Categorical Modelling of Logic Programming

Coalgebra, Functorial Semantics, String Diagrams

Tao Gu

University College London
Department of Computer Science

A dissertation submitted in partial fulfillment of the requirement for the degree of Doctor of Philosophy in Computer Science of University College London, October 2022

I, Tao Gu, confirm that the work presented in this thesis is my own. Where information has been derived from other sources, I confirm that this has been indicated in the thesis.

Abstract

Logic programming (LP) is driven by the idea that logic subsumes computation. Over the past 50 years, along with the emergence of numerous logic systems, LP has also grown into a large family, the members of which are designed to deal with various computation scenarios. Among them, we focus on two of the most influential quantitative variants are probabilistic logic programming (PLP) and weighted logic programming (WLP).

In this thesis, we investigate a uniform understanding of logic programming and its quantitative variants from the perspective of category theory. In particular, we explore both a coalgebraic and an algebraic understanding of LP, PLP and WLP.

On the coalgebraic side, we propose a goal-directed strategy for calculating the probabilities and weights of atoms in PLP and WLP programs, respectively. We then develop a coalgebraic semantics for PLP and WLP, built on existing coalgebraic semantics for LP. By choosing the appropriate functors representing probabilistic and weighted computation, such coalgeraic semantics characterise exactly the goal-directed behaviour of PLP and WLP programs.

On the algebraic side, we define a functorial semantics of LP, PLP, and WLP, such that they three share the same syntactic categories of string diagrams, and differ regarding to the semantic categories according to their data/computation type. This allows for a uniform diagrammatic expression for certain semantic constructs. Moreover, based on similar approaches to Bayesian networks, this provides a framework to formalise the connection between PLP and Bayesian networks. Furthermore, we prove a sound and complete aximatization of the semantic category for LP, in terms of string diagrams. Together with the diagrammatic presentation of the fixed point semantics, one obtain a decidable calculus for proving the equivalence between propositional definite logic programs.

Impact Statement

Inside Academia. Logic programming, probabilistic logic programming and weighted logic programming have been put into uniform categorical frameworks. We formalise the intuitive similarity between these variants of logic programming. We show that coalgebraic semantics can not only encode the goal-directed search space, but also capture the goal-directed calculation space for calculating the probabilities or weights in quantitative variants of logic programming. We propose a functorial semantics for the algebraic interpretation of LP, PLP and WLP, which feature in sharing the same syntactic category of string diagrams. This framework is expected to be generalised to other variants of logic programming, and to set the foundation of learning based on logic programs. Also, it opens the possibility to formally compare those logic programming paradigms with other computation models. Based on this functorial semantics, we give a complete diagrammatic calculus to turn the problem of equivalence between propositional definite logic programs into proving algebra equations, and indicate the potential generalisation to probabilistic and weighted setting. The results in this thesis have been published in several conferences [GZ19, GZ21, GPZ22] and in one journal article [ZG21].

Outside Academia. The uniform categorical framework in this thesis provides a guiding principle for designing new types of logic programming. For instance, since in our functorial semantics, PLP and WLP alternate the semantic categories of LP in two different ways, it indicates that a probabilistic version of weighted logic programming is in principle well-defined. In the functorial semantics for WLP, the defining property of the semantic category to be traced provides a general criterion for the choice of the underlying semiring of WLP for the programs to have well-defined semantics.

Also, the complete diagrammatic calculus for propositional definite logic programs indicates a potential alternative for program verification in terms of diagrams. While we deal with very simple logic programs here, one can potentially generalise to a graphical calculus for various Kleene algebra, used for verifying program equivalence.

Contents

Chapter 1

Introduction

Logic programming (LP) is among the first attempts towards knowledge representation in artificial intelligence, whose history can be traced back to the 1970s. In its simplest version, LP is built upon the observation that the seemingly simple logic of Horn clauses is powerful enough to encode many real-world reasoning scenarios and at the same time enjoys feasible algorithms for proof-search.

Logic programming, of course, is not the only contestant. Yet there are two key features, as Kowalski [Kow88] stated, that distinguish LP from mechanical theorem-proving paradigms. First, "(i)t exploits the fact that logic can be used to express definitions of computable functions and procedures". Second, "it exploits the use of proof procedures that perform deductions in a goal-directed manner, to run such definitions as programs". These two features induce the *declarative* and *procedural* understanding of knowledge, respectively. For instance, think about the simple Horn clause $\mathtt{path}(\mathtt{x},\mathtt{y}) \leftarrow \mathtt{path}(\mathtt{x},\mathtt{z}), \mathtt{edge}(\mathtt{z},\mathtt{y})$. We can have (at least) two readings of this clause—from right to left and from left to right—that correspond to the aforementioned declarative and procedural perspectives: (1) if there is a path from $\mathtt{x}$ to $\mathtt{z}$ and an edge $\mathtt{z} \to \mathtt{y}$, then there is also a path from $\mathtt{x}$ to $\mathtt{y}$; (2) to show that there is a path from $\mathtt{x}$ to $\mathtt{y}$, one has to prove that there exists some $\mathtt{z}$ such that there is a path from $\mathtt{x}$ to $\mathtt{z}$, and $\mathtt{x} \to \mathtt{z}$ is an edge. From a logic point of view, these two features corresponds to the *deductive* and *reductive* reading of proof rules: one can read on rule as either that certain assumptions entails the conclusion, or that to prove certain conclusion one has to prove these assumptions.

The approach we take in this thesis is to rephrase these two perspectives in logic programming as *algebraic* and *coalgebraic*, respectively. On one hand, taking the algebraic perspective, semantics of logic programs are defined as operations over a suitable algebra domain. On the other hand, from the coalgbraic point of view, a logic program can be viewed as a system whose unfolding corresponds to the reduction of a given goal to a set of subgoals. Unsurprisingly, the algebraic perspective has been explored since the early age of LP [Sco70], while the coalgebraic perspective receives a relatively late formal treatment [KMP10], even though the underlying goal-reduction methodology can be traced back to the early days of LP.

It is common practice in theoretical computer science that, after a classical model is invented, applied and stabilised, it is then generalised to deal with more complex scenarios. Let us take automata as an example. Automata has grown from (non)-deterministic finite automata where each state either accepts or rejects an input string into a giant family, including but not restricted to: probabilistic automata where at every state there is a probability that it accepts a given string [Rab63]; weighted automata in which every state assigns weights to the input strings [Sch61]; nominal automata whose infinite behaviour can be encoded finitely using theory of nominal sets [LKB14].

Similar evolution takes place in the world of logic programming. As a programming formalism, LP has been extended not only regarding the underlying logic (e.g. logic with negation [AB94], linear logic [Mil04]) but also in the data type to adapt to non-classical circumstances. Two of the most prominent variants in the latter sense are *probabilistic* (PLP) and *weighted* logic programming (WLP). In a nutshell, PLP deals with scenarios with random events and answers the question of the probability of certain event; WLP generalises from Boolean values to a wide class of semirings, and instead of the truth value of atoms it calculates the weights of atoms. Consequently, the application of logic programming has also been extended from automated theorem-proving and knowledge representation to algorithm specification [EGS05], statistic relational reasoning [DRKT07a, RKNP16], machine learning [MDR94, MDK$^+$18], etc.

However, such an expansion of logic programming is not paired with a unifying mathematical account. In particular, despite the seemingly close connection between logic programming and its quantitative relatives, there is a lack of formal investigation into such similarity, in particular using category theory. Category theory, as a universal language of mathematics, is particularly useful to describe such scenario as a unifying framework in which different types of systems are modelled by taking suitable categorical components. Coming back to our example of automata, variaties of automata can be uniformly modelled as coalgebras over its state space S as $S \to \mathcal{T}(S)$, where the functor $\mathcal{T}$ is determined by the *type* of the automata, such as (non-)deterministic [AM75b, AM75a], probabilistic [HJS07], nominal [MP98, LKB14]. Such a type is then used to define notions of behavioural equivalence and semantics in a uniform way.

Turning back to LP, the aim is to establish such a framework underlying classical, probabilistic and weighted logic programming such that their difference are both captured modularly as the change of certain categorical constructs. The categorical semantics for logic programs per se has been studied in various contexts. The categorical treatment of first-order logic as Lawvere theory [Law68] can be seen as preliminary work towards categorical understanding of logic programming. The key notion of *most general unifier* (mgu) in first-order logic programming has been charaterised as pullbacks [BMR01, ALM09]. Kinoshita and Power [KP96] proposed a typed logic programming language using indexed categories, such that models of the programs are captured as structure-preserving indexed functors from the indexed category of program to the indexed category of model. Proof-search in logic programs has been first treated coalgebraically in Komendantskaya et al. [KMP10], and later extended to first-order cases [KMP10, BZ13]. Yet to the best of our knowledge these work do not cover any variant of

LP.

This thesis aims at filling this gap, by presenting a categorical understanding of logic programming together with its probabilistic and weighted relatives. In particular, we explore both the algebraic and coalgebraic perspectives, since both the deductive and reductive readings of clauses remain in PLP and WLP. For instance, think of the clause $\mathtt{path}(\mathtt{x},\mathtt{y}) \leftarrow \mathtt{path}(\mathtt{x},\mathtt{z}), \mathtt{edge}(\mathtt{z},\mathtt{y})$ but now ornamented with a probability value 0.9. Then the clause $0.9 :: \mathtt{path}(\mathtt{x},\mathtt{y}) \leftarrow \mathtt{path}(\mathtt{x},\mathtt{z}), \mathtt{edge}(\mathtt{z},\mathtt{y})$ also has two explanations given the chosen perspective: (1) if there is a path from `x` to `z` and an edge $\mathtt{z} \to \mathtt{y}$, then there is probability 0.9 that there is also a path from `x` to `z`; (2) to calculate the probability that there is a path from `x` to `z`, one may first calculate that for the existence of a path from `x` to some `z` and the existence of a edge $\mathtt{z} \to \mathtt{y}$, and then multiply with 0.9. Let us continue with more details about the specific tools we adopt for unifying categorical treatment in both perspectives.

Coalgebraic perspective. To put it simple, coalgebras capture the behaviour of a dynamic system with state space S as certain function $\alpha\colon S \to \mathcal{T}(S)$ defining the observation and transition of states. The application of the coalgebra α on some state s can be seen as the one-step behaviour of the system at state s, intuitively described the unfolding of the system at state s. If such unfolding continues for infinitely many steps, the result is then essentially all the possible behaviour of the system starting from state s. Under some conditions such a space of behaviour lies in the final $\mathcal{T}$-coalgebra, and all possible behaviour at a specific state s is retrieved via the universal mapping property of the final coalgebra. This is the standard construction of the final coalgebra semantics.

In terms of logic and proofs, such unfolding matches the one-step reduction of a conclusion awaits to be proven to the set of assumptions that are sufficient to prove this conclusion. Consequently, the corresponding final coalgebra captures the search space for proof-searching in the system. Coalgebraic semantics for logic programming has been well studied. The treatment for propositional logic programs was developed in [KMP10, KP11]. The first-order case has been looked at from two perspectives [KP18]: the lax semantics [KPS16], and the saturated semantics [BZ13]. Both adopt Lawvere theories to deal with the variables, and differ in the treatment with naturality. However, these work has not been adapted to variations of logic programs. We follow the saturation semantics approach as it encodes both the theorem-proving and proof-searching perspectives of logic programs, and we provide a modular solution to the coalgebraic semantics of PLP and WLP where appropriate functors are chosen for the specific semantics of the programs.

We follow this line of research to provide a coalgebraic semantics for PLP and WLP that match the aforementioned reductive reading of clauses. In particular, a PLP (resp. WLP) program is viewed as a system, in which the states are atoms whose probabilities (resp. weights) await to be calculated; the transition of states amounts the reduction of the calculation of an atom A at the current state to that of the probabilities (resp. weights) of those atoms

$B_1, \ldots, B_k$ that deduce A, according to clauses of the form $p :: A \leftarrow B_1, \ldots, B_k$. Moreover, we choose to model PLP and WLP uniformly such that their connection with LP can be reflected from their coalgebra types. The coalgebraic semantics then encode the top-down computation of probabilities/weights, a generalisation of the search space in classical LP, using the final coalgebra.

Algebraic perspective. We adopt string diagrams and functorial semantics as the category theoretic tools to define a unifying algebraic semantics for the variants of logic programming. Functorial semantics was proposed by Lawvere [Law63] to study the model theory of algebras by separating the syntax and semantics of the algebraic theories. In a nutshell, the syntax of an algebraic theory is encoded as a category of terms called Lawvere theory; the semantic category corresponds to the models of the algebraic theory, e.g. Set for classical set-theoretical models. Models are identified with structure-preserving functors from the syntactic category of Lawvere theory to the semantic category Set. Homomorphisms between two models of a given algebraic theory are then natural transformations between their corresponding functors. Functorial semantics thus provides a uniform framework for studying algebras, where 'algebraic categories' become the object of interest, replacing the classical definition of equational definable classes of algebras. This open the possibility of an algebraic treatment of subjects beyond the traditional universal algebra domain.

For (propositional) logic programs, our main insight is that LP, PLP and WLP programs can share the same presentation while their difference in the data types is reflected at the level of the semantic categories in which they are interpreted. Specifically, we choose *string diagrams* as the syntax for logic programs. String diagrams as an intuitive and mathematical rigorous tool has been successfully applied in signal flow diagram [BSZ14, BSZ15], control theory [FSR16], quantum processes [AC09], synthetic statistics [Fri20], concurrency [BHP$^+$19], etc.

Besides the intuitive pictorial presentation, using string diagrams as syntax provide much advantages. First, the two-dimensional structure of string diagrams allows for more flexibility compared to traditional algebra terms. Second, string diagrams provide a uniform language to compare different computation models. In our case, we explore the connection between PLP and Bayesian networks, a probabilistic graphical model encoding conditional (in)dependence via directed acyclic graphs. While such mutual transformation between (certain subclasses of) Bayesian networks and PLP has been studied before [MSB08], the translation therein mingle the syntactic and semantic constructions. Instead, we characterise the transformation under the functorial semantics. As a result, we can clearly identify the roles played by syntax and semantics in the construction. Third, there are rich mathematical theories about string diagrams in the context of monoidal categories. For instance we will use *traces* to model feedback in logic program semantics. We also provide a complete diagrammatic calculus to reason about the equivalence of propositional definite logic programs.

Roadmap and original contributions. We outline here the structure of the thesis and the main original contribution of each chapter.

In Chapter 2 we recall the essential background knowledge for this thesis. The first part summarises the basics of classical, probabilistic and weighted logic programming. In the rest of this chapter we recall the basic theory of monoidal categories, with a focus on the graphical presentation as string diagrams. We demonstrate the power of diagrammatic calculus by recalling the theory of interacting Hopf algebra for linear relations.

In Chapter 3 we explore the coalgebraic semantics for PLP and WLP.

- We characterise propositional PLP and WLP programs as certain coalgebras with finite state spaces, in the form of isomorphisms between categories (Proposition 3.1.4, Proposition 3.3.3).
- We give two coalgebraic semantics for PLP programs, both of which are induced by the universal mapping property of the final coalgebras. The first semantics (Definition 3.1.33, Proposition 3.1.24) encodes the searching space of the proof for a goal, and the second semantics (Definition 3.1.33, Proposition 3.1.34) is the foundation for computing the probabilities of goals.
- We propose an algorithm to compute the distribution semantics based on the second coalgebraic semantics of a given goal in a PLP program (Proposition 3.2.23).
- Similarly, we provide a coalgebraic semantics for WLP programs (Definition 3.3.6) that captures the goal-directed behaviour of WLP (Proposition 3.3.7), and showcase how to derive the weighted of atoms under such coalgebraic semantics.

This chapter is based on the following papers.

– T. Gu, F. Zanasi. *A coalgebraic perspective on probabilistic logic programming.* CALCO 2019.

– F. Zanasi, T. Gu. *Coalgebraic semantics for probabilistic logic programming.* Logical Methods in Computer Science 17 (2021).

In Chapter 4 we develop a functorial semantics of logic programs.

- We propose a syntactic category of string diagrams for propositional LP, PLP and WLP, in a uniform fashion (Definition 4.1.4).
- We characterise classical (resp. probabilistic, weighted) logic programs as structure-preserving functors from the syntactic categories to the semantics categories, in the sense that there is one-one correspondence between programs and such functors (Proposition 4.2.5, 4.3.6, 4.4.6).

- We extend the string diagrams in the syntactic categories with feedback wires, and change the semantic categories into traced ones to interpret feedbacks. Based on this, we show that a least fixed point semantics of propositional definite logic programs and propositional weighted logic programs can be expressed diagrammatically (Proposition 4.2.17, Corollary 4.2.18).

- For acyclic probabilistic logic programs we express their canonical distribution semantics of atoms as string diagrams, in the sense that their interpretation under the functorial semantics retrieves exactly the probabilities of the atoms (Proposition 4.3.13).

- The mutual translation between propositional probabilistic logic programs and Boolean-valued Bayesian networks is explored under the functorial semantics. In particular, we show both directions of this translation can be characterised purely at the level of syntactic categories (Proposition 4.3.21, 4.3.28).

This chapter is based on the following paper.

– T. Gu, F. Zanasi. *Functorial semantics as a unifying perspective on logic programming.* CALCO 2021.

In 5 we develop a diagrammatic calculus using string diagrams to reason about propositional definite logic programs.

- We provide a sound and complete diagrammatic calculus for the inequational theory of monotone relations over finite Boolean algebras (Theorem 5.5.21).

- Along the way we consider a sub-theory, the inequational theory of monotone functions over finite Boolean algebras. We give a sound and complete string diagramsatic axiomatization of this theory (Theorem 5.4.5).

- We show that the diagrammatic calculus express the least fixed point semantics of definite logic programs (Proposition 5.6.4).

- We apply the diagrammatic calculus to give an axiomatisation to decide equality of programs: two programs are equal precisely if their corresponding diagrams can be proven to be equivalent in the inequational theory (Theorem 5.6.8).

This chapter is based on the following paper.

– T. Gu, R. Piedeleu, F. Zanasi. *A complete diagrammatic calculus for satisfiability.* MFPS 2022.

First there is an immediate followup regarding the two topics of this thesis. Given the coalgebraic and algebraic treatment of logic programming, one natural question is about their interaction. To the best of our knowledge, such interaction for LP is encoded as a bialgebraic semantics for definite logic programs [BZ15], where the algebraic and coalgebraic characterisation of a system interact via a distributive law [Kli11]. To explore a similar approach for PLP and WLP, the first technical challenge would be to find the appropriate functors to encode the algebraic and coalgebraic types of the programs, such that there exists a suitable distributive law. This problem is non-trivial for distribution monads and multiset functors [Var03, VW06, Jac21]. We are not devoted to this problem in the thesis and leave it for future work, since our algebraic semantics is defined in terms of functors, and it is not immediate to see how it interacts with the coalgebraic semantics.

Concerning the diagrammatic calculus, one immediate next step is to generalise the current calculus for definite logic programs to probabilistic and weighted case. For the former, we may consider a framework in which the diagrammatic language are equipped with discrete distributions over sub-diagrams, and the semantic category has morphisms that are discrete distributions of morphisms of that for logic programs.

For the latter, one may start with looking at some specific semiring structure, such as the min-plus semiring for shortest path problem.

Chapter 2

Background

2.1 Logic programming and its variants

In this part we recall logic programming as well as its probabilistic and weighted variants. We start with fixing some standard notation in predicate and propositional logic.

Definition 2.1.1. A *signature* Σ is a set of function symbols, each equipped with a fixed finite arity, explicitly expressed as the superscript when necessary. A *context* is a list of variables of the form $[x_1, x_2, \ldots, x_n]$. Given a signature Σ and a countably infinite set of variables $\mathit{Var} = \{x_1, x_2, \ldots, \}$, the Σ*-terms in context* $[x_1, \ldots, x_n]$ are defined inductively as follows:

- each $x_i \in \{x_1, \ldots, x_n\}$ is a Σ-term in context $[x_1, \ldots, x_n]$;
- for each $f^n \in \Sigma$, and Σ-terms $t_1, \ldots, t_n$ in context $[x_1, \ldots, x_n]$, $f(t_1, \ldots, t_n)$ is also a Σ-term in context $[x_1, \ldots, x_n]$.

A term is *ground* if it is variable-free.

With some abuse of notation, we shall often use n to denote the context $[x_1, \ldots, x_n]$. A Σ-term is simply a Σ-term in some context. We say a Σ-term t is *compatible* with context n if t is a Σ-term in context n. Note that a term t is compatible with multiple contexts (indeed contably many).

Definition 2.1.2. An *alphabet* $\mathcal{A}$ consists of a signature Σ, a set of variables Var, and a set of predicate symbols $\mathit{Pred} = \{P_1, P_2, \ldots\}$ (each with a fixed arity). Given an n-ary predicate symbol $P \in \mathit{Pred}$, and Σ-terms $t_1, \ldots, t_n$ in context n, $P(t_1 \cdots t_n)$ is called an *atom* over $\mathcal{A}$ in context n. We use $A, B, \ldots$ to denote atoms.

A literal is either an atom or a negated atom of the form $\neg A$ where A is an atom.

Definition 2.1.3. A *substitution* $\theta \colon n \to m$ is an n-tuple of Σ-terms in contexts m, namely $(t_1, \ldots, t_m)$. Given an atom A in context n, and a substitution $\theta = \langle t_1, \ldots, t_n \rangle \colon n \to m$, we write $A\theta$ for *substitution instance* of A obtained by replacing each appearance of x_i with t_i in A, namely $A[x_1/t_1, \ldots, x_n/t_n]$.

For convenience, we also use $\{B_1, \dots, B_k\}\theta$ as a shorthand for $\{B_1\theta, \dots, B_k\theta\}$.

Definition 2.1.4. Given two atoms A and B, a *unifier* of A and B is a pair $\langle \sigma, \tau \rangle$ of substitutions such that $A\sigma = B\tau$. *Term matching* is a special case of unification, where σ is the identity substitution. In this case we say that τ matches B with A if $A = B\tau$.

2.1.1 Logic programming

We now recall the basics of logic programming, and refer the reader to [Llo87] for a more systematic exposition.

Definition 2.1.5. A *general logic program* $\mathbb{L}$ consists of a finite set of clauses φ in the form $H \leftarrow L_1, \dots, L_k$, where H is an atom and $L_1, \dots, L_k$ are literals. H is called the *head* of φ (denoted as $\mathsf{head}(\varphi)$), and $L_1, \dots, L_k$ form the *body* of φ (denoted as $\mathsf{body}(\varphi)$). If the body of certain clause only contains positive literals, then it is a *definite* clause. A *definite* logic program consists of definite clauses.

We will mostly consider definite logic programs, as they have a more intuitive semantics defined in terms of least fixed-points. In the remainder of this thesis, by 'logic programs' we refer to definite logic programs unless otherwise stated.

When an atom awaits to be proven, it is sometimes referred to as a *goal*. Since one can regard a clause $H \leftarrow B_1, \dots, B_k$ as the logic formula $B_1 \wedge \cdots \wedge B_k \rightarrow H$, we say that a goal G is *derivable* in $\mathbb{L}$ if there exists a finite derivation of G with empty assumption using the clauses in $\mathbb{L}$.

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

Such goal-proving perspective is referred to as the operational semantics of logic programs. We focus on the denotational semantics of logic programming. Fix a logic program $\mathbb{L}$ over an alphabet $\mathcal{A}$.

Definition 2.1.6. The *Herbrand universe* $\mathcal{H}$ is the set of all ground Σ-terms. The *Herbrand base* At is the set of all atoms over $\mathcal{A}$. An *interpretation* $\mathcal{I}$ is simply a subset of At.

Definition 2.1.7. Every logic program $\mathbb{L}$ determines a monotone function called *immediate consequence operator* $\mathbf{T}_{\mathbb{L}} \colon \mathcal{P}(At) \rightarrow \mathcal{P}(At)$, such that for each $X \in \mathcal{P}(At)$, $\mathbf{T}_{\mathbb{L}}(X)$ is the set of all $A \in At$ such that there exists $\mathbb{L}$-clause φ and substitution θ such that $\mathsf{head}(\varphi)\theta = A$ and $\mathsf{body}(\varphi)\theta \subseteq X$. The *least Herbrand model* $\mathcal{M}^H_{\mathbb{L}}$ of $\mathbb{L}$ is the least fixed point of $\mathbf{T}_{\mathbb{L}}$.

Intuitively, $\mathbf{T}_{\mathbb{L}}$ maps an interpretation $\mathcal{I} \in \mathcal{P}(At)$ to the set of all atoms that are derivable from atoms in $\mathcal{I}$ using $\mathbb{L}$-clauses in one step. It is obviously monotone since the clauses are definite, and the larger X is, the more atoms one could derive from X using $\mathbb{L}$. Since $\mathbf{T}_{\mathbb{L}}$ is monotone on $\langle \mathcal{P}(At), \subseteq \rangle$, $\mathcal{M}^H_{\mathbb{L}}$ is well-defined as its least fixed point.

Definition 2.1.8. The *consequence operator* $\mathbf{C}_{\mathbb{L}}\colon \mathcal{P}(At) \to \mathcal{P}(At)$ maps $X \in At$ to the least Herbrand model of $\mathbb{L} \cup \{A \leftarrow . \mid A \in X\}$.

The consequence operator captures the algebraic reading of a logic program $\mathbb{L}$ as a 'black box': one can input atoms to $\mathbb{L}$, and $\mathbb{L}$ will output all the atoms provable using $\mathbb{L}$ and input atoms as facts.

We point out that, using the isomorphism $\mathcal{P}(X) \cong \mathbb{B}^X$ for arbitrary set X—where $\mathbb{B}$ is the two-element Boolean algebra $\{0, 1\}$ and $\mathbb{B}^X$ denotes the set of all functions $X \to \mathbb{B}$—all the above construction of the denotational semantics for definite logic programs can be restated in Boolean algebra terms. While the set-theoretical narrative is more standard, in the subsequent sections we will often adopt this perspective whenever necessary. For instance, we shall say an interpretation $\mathbf{T}_{\mathbb{L}}$ is a monotone function on $\mathbb{B}^{At}$. We shall also identify interpretations $\mathcal{I} \in \mathcal{P}(At)$ with states $s \in \mathbb{B}^{At}$, such that $s(A) = 1$ if and only if $A \in \mathcal{I}$, for arbitrary atom $A \in At$. A state $s \in \mathbb{B}^{At}$ *satisfies* an atom A if $s(A) = 1$, and *satisfies* a negated atom $\neg B$ if $s(B) = 0$. That s satisfies a literal L is denoted as $s \vDash L$.

2.1.2 Probabilistic logic programming

We now recall the basics of PLP; the reader may consult [DRKT07b, DRKT07a] for a more comprehensive introduction.

Definition 2.1.9. A *probabilistic clause* φ (or PLP clause) is of the form $r :: A \leftarrow L_1, \ldots, L_k$, where $r \in (0, 1]$ and $A \leftarrow L_1, \ldots, L_k$ is a logic program. The number r is referred to as $lab(\varphi)$. A *probabilistic logic program* (or PLP program) is a finite set of probabilistic clauses such that no two clauses share the same underlying logic program clause.

Sticking with the terminology for logic programming, we say a PLP clause is definite if no negation appears in its body, and a PLP program consisting of definite clauses is a definite program. A program $\mathbb{P}$ is *ground* if it does not have variables.

Example 2.1.10. As our leading example we introduce the following probabilistic logic program $\mathbb{P}_{al}$. It models the scenario of Mary's house alarm, which is supposed to detect burglars, but it may be accidentally triggered by an earthquake. Mary may hear the alarm if she is awake, but even if the alarm is not sounding, in case she experiences an auditory hallucination (paracusia). The language of $\mathbb{P}_{al}$ includes 0-ary predicates `Alarm`, `Eearthquake`, `Burglary`, and 1-ary predicates `Wake`$(-)$, `Hear_alarm`$(-)$ and `Paracusia`$(-)$, and signature $\Sigma = \{\texttt{Mary}^0\}$ consisting of a constant. We do not have variables here, so $\mathbb{P}_{al}$ is a ground program. For readability we abbreviate `Mary` as `M` in the program.

$$\begin{array}{rll|rll}
0.01 :: & \texttt{Earthquake} & \leftarrow & 0.01 :: & \texttt{Paracusia(M)} & \leftarrow \\
0.2 :: & \texttt{Burglary} & \leftarrow & 0.6 :: & \texttt{Wake(M)} & \leftarrow \\
0.5 :: & \texttt{Alarm} & \leftarrow \texttt{Earthquake} & 0.8 :: & \texttt{Hear\_alarm(M)} & \leftarrow \texttt{Alarm, Wake(M)} \\
0.9 :: & \texttt{Alarm} & \leftarrow \texttt{Burglary} & 0.3 :: & \texttt{Hear\_alarm(M)} & \leftarrow \texttt{Paracusia(M)}
\end{array}$$

As a generalisation of the pure case, in probabilistic logic programming one is interested in the *probability* of a goal G being provable in a program $\mathbb{P}$. There are potentially multiple ways to define such probability— in this paper we focus on the *distribution semantics* [DRKT07b].

The distribution semantics is defined paraterised on a given semantics or notion of derivability — say some $\vdash$ — with $\mathbb{L} \vdash A$ meaning 'atom A is true/derivable in the logic program $\mathbb{L}$'. Given a probabilistic logic program $\mathbb{P} = \{p_1 :: \varphi_1, \dots, p_n :: \varphi_n\}$ where each φ is a logic program clause, let $|\mathbb{P}|$ be its underlying pure logic program, namely $|\mathbb{P}| = \{\varphi_1, \dots, \varphi_n\}$. A *sub-program* $\mathbb{L}$ of $|\mathbb{P}|$ is a logic program consisting of a subset of the clauses in $|\mathbb{P}|$. This justifies using $\mathcal{P}(|\mathbb{P}|)$ to denote the set of all sub-programs of $|\mathbb{P}|$, and using $\mathbb{L} \subseteq |\mathbb{P}|$ to denote that $\mathbb{L}$ is a sub-program of $\mathbb{P}$. The central concept of the distribution semantics is that $\mathbb{P}$ determines a distribution $\mu_{\mathbb{P}}$ over the sub-programs $\mathcal{P}(|\mathbb{P}|)$: for any $\mathbb{L} \in \mathcal{P}(|\mathbb{P}|)$, $\mu_{\mathbb{P}}(\mathbb{L}) \coloneqq \prod_{\varphi_i \in \mathbb{L}} p_i \prod_{\varphi_j \in |\mathbb{P}| \setminus \mathbb{L}} (1 - p_j)$. The value $\mu_{\mathbb{P}}(\mathbb{L})$ is called the *probability* of the sub-program $\mathbb{L}$.

Definition 2.1.11. The *success probability* $Pr_{\mathbb{P}}(G)$ of a goal G in program $\mathbb{P}$ is the sum of the probabilities of all the sub-programs of $\mathbb{P}$ in which G is true/derivable:

$$Pr_{\mathbb{P}}(G) \coloneqq \sum_{|\mathbb{P}| \supseteq \mathbb{L} \vdash G} \mu_{\mathbb{P}}(\mathbb{L}) = \sum_{|\mathbb{P}| \supseteq \mathbb{L} \vdash G} \left(\prod_{\varphi_i \in \mathbb{L}} p_i \cdot \prod_{\varphi_j \in |\mathbb{P}| \setminus \mathbb{L}} (1 - p_j) \right) \tag{2.1}$$

Intuitively, under the distribution semantics every clause in $\mathbb{P}$ is regarded as a random event, then every sub-program $\mathbb{L}$ can be seen as a possible world, and $\mu_{\mathbb{P}}$ is a distribution over the possible worlds. The success probability of G is then simply the sum of the probabilities of all worlds in which G is true or derivable, depending on the fixed semantics of the underlying logic programs.

One comment regarding the execution, while the naive computation of such probability is to enumerate all sub-programs $L \subseteq |\mathbb{P}|$, more effective implementation have been invented, for example an algorithm based on BDD tree is introduced in Kimmig et al. [KDDR$^+$10].

Example 2.1.12. Recall the (definite) PLP program $\mathbb{P}_{al}$ from Example 2.1.10, and consider the least Herbrand semantics of definite logic program. For the goal `Hear_alarm(M)`, we can compute its success probability $Pr_{\mathbb{P}_{al}}$(`Hear_alarm(M)`) using (2.1), and the result is 0.091102896. For instance, the subprogram of $|\mathbb{P}_{al}|$ consisting of `Earthquake` ← ., `Alarm` ← `Earthquake`., `Wake(M)` ← ., `Hear_alarm(M)` ← `Alarm`, `Wake(M)`. has `Hear_alarm(M)` in its least Herbrand model, and it has probability under $\mu_{\mathbb{P}_{al}}$ is $0.01 \times (1-0.2) \times 0.5 \times (1-0.9) \times (1-0.01) \times 0.6 \times 0.8 \times (1-0.3)$, which contributes to $Pr_{\mathbb{P}_{al}}$(`Hear_alarm(M)`).

We recall *Bayesian networks*, a probabilistic graphical model closely related to (ground) PLP programs.

Definition 2.1.13. A *Bayesian network* $\mathcal{B}$ consists of:

- a directed acyclic graph (DAG) G whose nodes represents random variables;

- A value space for each random variable;
- A set of conditional probabilities $Pr_{\mathcal{B}}(X \mid pa(X))$, one for each node X, where $pa(X)$ is the set of parents of X in G.

Moreover, it satisfies the factorisation property, where $X_1, \dots, X_n$ are all the nodes in G, and $x_1, \dots, x_n$ are values of $X_1, \dots, X_n$, respectively:

$$Pr_{\mathcal{B}}(X_1 = x_1, \dots, X_n = x_n) = \prod_{i=1,\dots,n} Pr_{\mathcal{B}}(X_i = x_i \mid pa(X_i) = pa(x_i))$$

When the value space for each of the random variable is the $\mathbb{B}$, we say $\mathcal{B}$ is a *Boolean* Bayesian network. We postpone an example of Bayesian network till Example 4.3.19, where Bayesian networks are actually used in comparison with probabilistic logic programs.

2.1.3 Weighted logic programming

Finally we introduce the weighted variation of logic programming. Let us fix a semiring $\mathbf{K} = \langle \mathbf{K}, +, \cdot, \mathbf{0}, \mathbf{1} \rangle$, together with a complete lattice structure $\preceq$ that is *compatible* with the semiring structure: (i) $\forall x_1, x_2, y_1, y_2 \in \mathbf{K}$, if $x_1 \preceq y_1$ and $x_2 \preceq y_2$, then $x_1 + x_2 \preceq y_1 + y_2$ and $x_1 \cdot x_2 \preceq y_1 \cdot y_2$; (ii) $\mathbf{0}$ is the $\preceq$-least element in $\mathbf{K}$. We use the standard notation $\wedge$, $\top$, $\vee$, $\perp$ for the lattice structure. Thus $\mathbf{0} = \perp$.

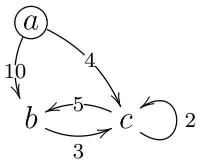
Example 2.1.14. The min-plus semiring $\mathbf{MinPlus} = \langle \mathbb{N} \cup \{+\infty\}, \min, +, \infty, 0 \rangle$ has the underlying set natural number equipped with the positive infinity. The operations min and $+$ are respectively minimise and plus, with $\min\{n, +\infty\} = n$ and $n + (+\infty) = +\infty$ for arbitrary $n \in \mathbb{N} \cup \{+\infty\}$. The semiring axioms are straightforward to verify. For instance, the distributive axioms holds: $\forall x, y, z \in \mathbb{N} \cup \{+\infty\}$, $x + \min\{y, z\} = \min\{x + y, x + z\}$.

The complete lattice structure $\preceq$ is the standard $\geq$ on $\mathbb{N}$ (note the direction) together with $\{(+\infty, n) \mid n \in \mathbb{N} \cup \{+\infty\}\}$. The join and meet are then min and max, respectively. It is complete because the least upper bound and greatest lower bound are both well-defined as min and max; in particular it is max-complete by including $+\infty$.

Definition 2.1.15. A *weighted clause* φ based on $\mathbf{K}$ is of the form $w :: A \leftarrow B_1, \dots, B_k$, where $A, B_1, \dots, B_k$ are atoms, and $w \in \mathbf{K}$. We refer to w, A, and $\{B_1, \dots, B_k\}$ as $lab(\varphi)$, $\mathsf{head}(\varphi)$ and $\mathsf{body}(\varphi)$, respectively. A *weighted logic program* (or simply *WLP program*) $\mathbb{W}$ is a finite set of weighted clauses such that no two clauses share the same heads and bodies.

Example 2.1.16. The min-plus semiring $\mathbb{N} \cup \{+\infty\}$ from Example 2.1.14 can be used for shortest path problems. Consider the WLP program $\mathbb{P}_{sp}$ consisting of all *the grounding of* the clauses below left in (2.2), which describes the reachability condition of the weighted directed graph D below right in (2.2). Then the answer to the shortest path from the initial state to a state $\mathtt{x}$ 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.

$$\begin{array}{rccl|rccl}
0 :: & \texttt{initial(a)} & \leftarrow & . & 10 :: & \texttt{edge(a,b)} & \leftarrow & . \\
4 :: & \texttt{edge(a,c)} & \leftarrow & . & 3 :: & \texttt{edge(b,c)} & \leftarrow & . \\
5 :: & \texttt{edge(c,b)} & \leftarrow & . & 2 :: & \texttt{edge(c,c)} & \leftarrow & . \\
0 :: & \texttt{reachable(x)} & \leftarrow & \multicolumn{5}{l}{\texttt{initial(x)}.} \\
0 :: & \texttt{reachable(x)} & \leftarrow & \multicolumn{5}{l}{\texttt{reachable(y)},\texttt{edge(y,x)}.}
\end{array} \tag{2.2}$$



Definition 2.1.17. Every weighted logic program $\mathbb{W}$ induces a monotone function $\mathbf{T}^w_{\mathbb{W}}\colon \mathbf{K}^{At} \to \mathbf{K}^{At}$ called *weighted immediate consequence operator*, such that for arbitrary $u \in \mathbf{K}^{At}$ and $A \in At$,

$$\mathbf{T}^w_{\mathbb{W}}(u)(A) = \sum_{\varphi\in\mathbb{W},\mathsf{head}(\varphi)=A} \left(lab(\varphi) \cdot \left(\prod_{B_i\in\mathsf{body}(\varphi)} u(B_i) \right) \right) \tag{2.3}$$

The *weighted consequence operator* $\mathbf{C}^w_{\mathbb{W}}\colon \mathbf{K}^{At} \to \mathbf{K}^{At}$ maps $u \in \mathbf{K}^{At}$ to the least fixed point of $\lambda v.u + \mathbf{T}^w_{\mathbb{W}}(v)$. The least model of $\mathbb{W}$, denoted as $\mathcal{M}^w_{\mathbb{W}}$, is the least fixed point of $\lambda v.\mathbf{T}^w_{\mathbb{W}}(v)$.

Both operators are monotone: $\mathbf{T}^w_{\mathbb{W}}$ is monotone because because it only involves $+$ and $\cdot$, which are monotone on $\langle \mathbf{K}, \preceq \rangle$ by assumption; $\mathbf{C}^w_{\mathbb{W}}$ is monotone because if $u_1, u_2 \in \mathbf{K}^{At}$ satisfy $u_1 \preceq u_2$, then for any $v \in \mathbf{K}^{At}$, $u_1 + \mathbf{T}^w_{\mathbb{W}}(v) \preceq u_2 + \mathbf{T}^w_{\mathbb{W}}(v)$, thus $\lambda v.u_1 + \mathbf{T}^w_{\mathbb{W}}(v) \preceq \lambda v.u_2 + \mathbf{T}^w_{\mathbb{W}}(v)$ Regarding the least model, $\mathcal{M}^w_{\mathbb{W}}$ is equivalently $\mathbf{C}^w_{\mathbb{W}}(\bar{\bot})$, where $\bar{\bot} \in \mathbf{K}^{At}$ is the constant $\bot$ state.

2.2 Basic category theory

We assume familiarity with basic category theory, such as the notions of categories, functors, natural transformation etc. These can be found in any standard textbook on category theory, such as MacLane [Mac71], Awodey [Awo10]. For the concepts that we recall in this section, we refer to the following literature for details.

Jacobs [Jac17] is a recent introductory book to coalgebras. Fong and Spivak [FS19] is an approachable introduction to monoidal categories and string diagrams. Zanasi [Zan15] contains a comprehensive discussion of the theory of PROPs (categories of **pro**ducts with **p**ermutation).

2.2.1 Monads and multiset functors

We recall some well-known functors that will be used in the later chapters. Some of them carry extra structure and form so-called *monads*.

Definition 2.2.1 (Monad)**.** A *monad* is an endofunctor $\mathcal{T}\colon \mathbb{C} \to \mathbb{C}$ equipped with a multiplication $\mu^{\mathcal{T}}\colon \mathcal{T}\mathcal{T} \Rightarrow \mathcal{T}$ and $\eta\colon \mathbf{1} \Rightarrow \mathcal{T}$ natural transformations commuting the following diagrams:

$$\begin{tikzcd}
\mathcal{T}\mathcal{T}\mathcal{T} \arrow[r, "\mathcal{T}\mu"] \arrow[d, "\mu\mathcal{T}"'] & \mathcal{T}\mathcal{T} \arrow[d, "\mu"] \\
\mathcal{T}\mathcal{T} \arrow[r, "\mu"] & \mathcal{T}
\end{tikzcd}
\qquad
\begin{tikzcd}
 & \mathcal{T} \arrow[dl, "\eta\mathcal{T}"'] \arrow[d, equal] \arrow[dr, "\mathcal{T}\eta"] & \\
\mathcal{T}\mathcal{T} \arrow[r, "\mu"] & \mathcal{T} & \mathcal{T}\mathcal{T} \arrow[l, "\mu"']
\end{tikzcd}$$

A closely related notion is that of adjunction.

Definition 2.2.2 (Adjunction)**.** Two functors $\mathcal{F}\colon \mathbb{C} \to \mathbb{D}$ and $\mathcal{G}\colon \mathbb{D} \to \mathbb{C}$ form an *adjunction* if there exist natural transformations $\eta\colon \mathbf{1}_{\mathbb{C}} \Rightarrow \mathcal{G} \circ \mathcal{F}$ and $\varepsilon\colon \mathcal{F} \circ \mathcal{G} \Rightarrow \mathbf{1}_{\mathbb{D}}$ (called the unit and counit of the adjunction) such that the following diagrams commute:

$$\begin{tikzcd}
\mathcal{F}\mathcal{G}\mathcal{F} \arrow[rr, equal] \arrow[dr, "\varepsilon\mathcal{F}"'] & & \mathcal{F}\mathcal{G}\mathcal{F} \\
 & \mathcal{F} \arrow[ur, "\mathcal{F}\eta"'] &
\end{tikzcd}
\qquad
\begin{tikzcd}
\mathcal{G}\mathcal{F}\mathcal{G} \arrow[rr, equal] \arrow[dr, "\mathcal{G}\varepsilon"'] & & \mathcal{G}\mathcal{F}\mathcal{G} \\
 & \mathcal{G} \arrow[ur, "\eta\mathcal{G}"'] &
\end{tikzcd}$$

In this case we say $\mathcal{F}$ is the left adjunction of $\mathcal{G}$, denoted as $\mathcal{F} \dashv \mathcal{G}$.

Every adjunction $\mathcal{F} \dashv \mathcal{G}$ induces a monad $\mathcal{G} \circ \mathcal{F}\colon \mathbb{C} \to \mathbb{C}$, whose monad unit is precisely the unit of the adjunction. We choose the specific definition of adjunction as in Definition 2.2.2 because it smoothly generalises to 2-categories. Indeed, the specific definition of adjunction in Definition 2.2.2 is that instantiated in the 2-category Cat of categories, functors, and natural transformation. We mention that the other direction is also true, namely every monad induces an adjunction. However not every adjunction arises from a monad, and those adjunctions that are indeed induced by monads are called monadic adjunction, see e.g. Mac Lane [Mac71, Chapter VI]. One such construction is via the Kleisli category.

Definition 2.2.3. Let $\mathcal{T}\colon \mathbb{C} \to \mathbb{C}$ be a monad. The *Kleisli category* $\mathcal{K}\ell(\mathbb{C})$ has $\mathbb{C}$-objects as objects; morphisms of type $X \rightarrowtriangle Y$ are $\mathbb{C}$-morphisms of the form $X \to \mathcal{T}Y$. The identity morphism $X \rightarrowtriangle X$ is the unit $\eta^{\mathcal{T}}_X\colon X \to \mathcal{T}X$. The composition of morphisms $f\colon X \rightarrowtriangle Y$ and $g\colon X \rightarrowtriangle Y$ is $g \odot f = X \xrightarrow{f} \mathcal{T}Y \xrightarrow{\mathcal{T}g} \mathcal{T}\mathcal{T}Y \xrightarrow{\mu^{\mathcal{T}}_Y} \mathcal{T}Y$.

The adjunction consists of $\mathcal{F}\colon \mathbb{C} \to \mathcal{K}\ell(\mathbb{C})$ mapping $f\colon X \to Y$ to $\eta_Y \circ f\colon X \rightarrowtriangle Y$ and $\mathcal{G}\colon \mathcal{K}\ell(\mathbb{C}) \to \mathbb{C}$ mapping $g\colon X \rightarrowtriangle Y$ to $\mu_Y \circ \mathcal{T}(g)\colon \mathcal{T}X \to \mathcal{T}Y$, with $\begin{tikzcd} \mathbb{C} \arrow[r, bend left, "\mathcal{F}"] \arrow[r, phantom, "\perp"] & \mathcal{K}\ell(\mathbb{C}) \arrow[l, bend left, "\mathcal{G}"] \end{tikzcd}$. A closely related construction is that of *Eilenburg-Moore categories*, indeed they can be seen as two extremal solution to the question of whether every monad arises from an adjunction.

Now we look at a wide class of functors on the category Set of sets that are induced by *multisets*. Under certain conditions these functors are indeed monads.

Definition 2.2.4. Given a monoid $\mathbf{S} = \langle S, +, \mathbf{0}\rangle$, a $\mathbf{S}$-*multiset over set* X is a function $\varphi\colon X \to S$ such that the set $\{x \in X \mid \varphi(x) \neq \mathbf{0}\}$ is finite.

We call $\{x \in X \mid \varphi(x) \neq \mathbf{0}\}$ the *support* of φ, denoted as $\mathsf{supp}(\varphi)$. So equivalently we say $\mathbf{S}$-multisets are those functions $\varphi\colon X \to S$ that have finite supports. We adopt the 'ket' notation to write multisets succinctly as a formal sum: if $\mathsf{supp}(\varphi) = \{x_1, \dots, x_m\}$, then we write φ as $\varphi(x_1)|x_1\rangle + \cdots + \varphi(x_m)|x_m\rangle$ or $\sum_{i=1}^{m} \varphi(x_i)|x_i\rangle$. Such 'ket' notation adopts the following conventions: $s_1|x_1\rangle + \mathbf{0}|x_2\rangle = s_1|x_1\rangle$, and $s_1|x\rangle + s_2|x\rangle = (s_1 + s_2)|x\rangle$.

Every monoid induces an endofunctor on Set, mapping each set X to all multisets over X.

Definition 2.2.5. Given a monoid $\mathbf{S} = \langle S, +, \mathbf{0}\rangle$, the *multiset functor* $\mathcal{M}_{\mathbf{S}} \colon \mathsf{Set} \to \mathsf{Set}$ is defined as follows:

- On objects, given an arbitrary $X \in \mathsf{ob}(\mathsf{Set})$, $\mathcal{M}_{\mathbf{S}}(X)$ is the set of all $\mathbf{S}$-multisets over X.
- On morphisms, given an arbitrary function $f \colon X \to Y$, $\mathcal{M}_{\mathbf{S}}(f)$ maps $\sum_{i=1}^{m} \varphi(x_i)|x_i\rangle$ to $\sum_{i=1}^{m} \varphi(x_i)|f(x_i)\rangle$.

Moreover, when the underlying monoid $\mathbf{S}$ has in addition semiring structure, then its induced multiset functor is indeed a monad. Recall that a semiring $\mathbf{K} = \langle K, +, \cdot, \mathbf{0}, \mathbf{1}\rangle$ has two monoid structure: (commutative) $\langle K, +, \mathbf{0}\rangle$ and $\langle K, \cdot, \mathbf{1}\rangle$ satisfying $x \cdot (y + z) = x \cdot y + x \cdot z$, $(x + y) \cdot z = x \cdot z + x \cdot z$, and $\mathbf{0} \cdot x = \mathbf{0}$.

Proposition 2.2.6 ([Jac17])**.** *Given a semiring* $\mathbf{K}$*, the multiset functor* $\mathcal{M}_{\mathbf{K}}$ *is a monad.*

It is helpful to spell out the unit and multiplication of the semiring monad. Given a set X, $\eta_X^{\mathcal{M}_{\mathbf{K}}} \colon X \to \mathcal{M}_{\mathbf{K}}(X)$ maps x to $\mathbf{1}|x\rangle$; $\mu_X^{\mathcal{M}_{\mathbf{K}}} \colon \mathcal{M}_{\mathbf{K}}\mathcal{M}_{\mathbf{K}}(X) \to \mathcal{M}_{\mathbf{K}}(X)$ maps $s_1|\varphi_1\rangle + \cdots + s_k|\varphi_k\rangle$ to $s_1 \cdot \varphi_1 + \cdots + s_k \cdot \varphi_k$.

Example 2.2.7. Here are some well-known multisets and their induced multiset functors.

1. Let $\mathbf{B} = \langle \{0, 1\}, \vee, \wedge, 0, 1\rangle$ be the Boolean semiring, then $\mathbf{B}$-multisets over a set X are functions $\varphi \colon X \to \{0, 1\}$ such that there are finitely many $x \in X$ satisfying $\varphi(x) = 1$. Given a function $f \colon X \to Y$ and $\mathbf{B}$-multiset φ over X, $\mathcal{M}_{\mathbf{B}}(\varphi)(y) = 1$ if and only if there exists some $x \in X$ such that $f(x) = y$ and $\varphi(x) = 1$.

   Note that $\mathbf{B}$-multisets over X are isomorphic to finite subsets of X: every φ induces a finite subset $A \subseteq X$ such that $x \in A$ if and only if $\varphi(x) = 1$, and vice versa. Such a 'set-theoretical view' of $\mathcal{M}_{\mathbf{B}}$ defines the *finite powerset monad* $\mathcal{P}_f \colon \mathsf{Set} \to \mathsf{Set}$. The monad structure is as follows: given a set X,

$$\begin{array}{ll} \eta_X^{\mathcal{P}_f} \colon X \to \mathcal{P}_f(X) & (x \in X) \mapsto \{x\} \\ \mu_X^{\mathcal{P}_f} \colon \mathcal{P}_f\mathcal{P}_f(X) \to \mathcal{P}_f(X) & (\mathcal{A} \subseteq \mathcal{P}_f(X)) \mapsto \bigcup \mathcal{A} \end{array}$$

2. Let $\mathbf{N} = \langle \mathbb{N}, +, 0\rangle$ be the natural number monoid, then $\mathbf{N}$-multisets are exactly finite multisets in the normal sense, or *bags*. Equipping it with $\times$ and 1 return a semiring, and thus induces a monad $\mathcal{M}_{\mathbf{N}}$.

3. Let $\mathbf{Pr} = \langle [0, 1], \vee_{pr}, 0\rangle$ be the probabilistic monoid where the probabilistic sum (*cf.* [WHK10]) is defined as $p \vee_{pr} q \coloneqq 1 - (1 - p) \cdot (1 - q)$ for arbitrary $p, q \in [0, 1]$. The intuition is

that, if p and q are the probabilities of two independent (Boolean) events A and B, then $p \vee_{pr} q$ is the probability of the event 'A or B'. Then **Pr**-multisets over a set X are functions $\varphi\colon X \to [0,1]$ that assigns a nonzero probability value to only finitely many elements in X. We denote the induced multiset functor as $\mathcal{M}_{pr}$. In particular, given a function $f\colon X \to Y$, the image of φ under $\mathcal{M}_{pr}(f)\colon \mathcal{M}_{pr}(X) \to \mathcal{M}_{pr}(Y)$ is the multiset over Y such that for arbitrary $y \in Y$, let $x_1, \ldots, x_k$ be all the $x \in X$ such that $f(x) = y$, $\mathcal{M}_{pr}(f)(\varphi)(y)$

4. Another important monad is the distribution monad $\mathcal{D}$. On objects, $\mathcal{D}(X)$ is the set of all finite probability distributions over X, namely $p_1|x_1\rangle + \cdots + p_k|x_k\rangle$ where $\sum_{i=1}^{k} p_i = 1$. On morphisms, $\mathcal{D}(f\colon X \to Y)$ maps a finite distribution $p_1|x_1\rangle + \cdots + p_k|x_k\rangle$ to $p_1|f(x_1)\rangle + \cdots + p_k|(x_k)\rangle$. Despite the similarity of the behaviour of $\mathcal{D}$ over morhpisms with that of multiset functors, $\mathcal{D}$ itself is *not* a multiset functor, yet can be seen as certain multiset functor plus a restriction of scalars. To be precise, consider the real number semiring $\mathbf{R} = \langle \mathbb{R}, +, \times, 0, 1\rangle$, which induces a monad. Then $\mathcal{D}(X)$ is the subset of $\mathcal{M}_{\mathbf{R}}(X)$ whose multisets are restricted to those with non-negative weighted and sum up to 1. A closely related monad is the sub-distribution monad $\mathcal{D}_{\leq}$ where the condition $\sum_{i=1}^{k} p_i = 1$ is loosen to $\sum_{i=1}^{k} p_i \leq 1$.

5. Let $\mathbf{MinPlus} = \langle \mathbb{N} \cup \{+\infty\}, \min, +, +\infty, 0\rangle$ be the *min-plus semiring*, where $\mathbb{N} \cup \{+\infty\}$ is adding $+\infty$ to the set of natural numbers as the unit of min. Then **MinPlus**-multisets over a set X are functions $\varphi\colon X \to \mathbb{N} \cup \{+\infty\}$ that assign a finite weight to only finitely many elements in X. Such tropical semiring is used in shortest-path problems, which calculates the least path between two given vertices in a weighted graph. The definition can be generalised to have rational numbers or real numbers as weights, and is known as *tropical semiring*, see e.g. Speyer [SS09] for tropical arithmetic.

There is an interesting connection between $\mathcal{M}_{pr}$ and $\mathcal{D}$. Intuitively, if one has finitely many independent random Boolean events (namely the outcome is either 0 or 1), then they induce a finite distribution over all the possible worlds in which they may happen or not. We leave the details to Proposition 3.1.32.

2.2.2 Coalgebra

Definition 2.2.8 (Coalgebra)**.** A *coalgebra* of type $\mathcal{F}$ where $\mathcal{F}$ is an endofunctor $\mathbb{C} \to \mathbb{C}$ is a $\mathbb{C}$-morphism of the form $\alpha\colon X \to \mathcal{F}X$.

The object X is sometimes refer to as the *base* of the coalgbera α. Fix an endofunctor $\mathcal{F}\colon \mathbb{C} \to \mathbb{C}$, all $\mathcal{F}$-coalgebras and coalgebra morphisms between them form a category $\mathbf{Coalg}(F)$, where a *coalgebra morphism* between two $\mathcal{F}$-coalgebras $\alpha\colon X \to \mathcal{F}X$ and $\beta\colon Y \to$

$\mathcal{F}Y$ is a $\mathbb{C}$-morphism $f\colon X \to Y$ such that the following diagram commutes:

$$\begin{tikzcd} X \arrow[r, "f"] \arrow[d, "\alpha"'] & Y \arrow[d, "\beta"] \\ \mathcal{F}X \arrow[r, "\mathcal{F}(f)"] & \mathcal{F}Y \end{tikzcd}$$

Coalgebras are useful to model transition systems, which we briefly recall the typical approach. Given a system S, the first step is to find a coalgebraical representation of its one-step or small-step behaviour as a $\mathcal{T}$-coalgebra for some functor $\mathcal{T}$. The $\mathcal{T}$-coalgebras form a category, whose objects are $\mathcal{T}$-coalgebras, and whose morphisms are morphisms between the underlying base of the coalgebras such that the square involving the coalgebras commute. If the functor $\mathcal{T}$ is good enough, the category of $\mathcal{T}$-coalgebras have final objects, called *final $\mathcal{T}$-coalgebras.* Then the coalgebraic semantics of system S is define as the universal morphism from its corrresponding $\mathcal{T}$-coalgebra to the final $\mathcal{T}$-coalgebra. Intuitively, the final $\mathcal{T}$-coalgebra contain exactly all the possible big-step behaviour of systems of type $\mathcal{T}$, and the universal morphism to the final $\mathcal{T}$-coalgebra assigns to a state its big-step behaviour in the system.

Example 2.2.9. Given a set of alphabet Σ, a deterministic finite automaton (DFA) is of the form $S = \langle Q, \delta, F\rangle$, where Q is a finite set of states, $\delta\colon Q \times \Sigma \to Q$ is a transition function, $F \subseteq Q$ is the set of final states. A state q *accepts* a finite string σ over Σ if the run of S starting from q along σ terminates at some state in F. Every DFA can be encoded as a coalgebra, whose coalgebra type is the Set-functor $(\cdot)^\Sigma \times \{0,1\}$. In particular, the DFA S is the coalgebra $\mathsf{s}\colon Q \to Q^\Sigma \times \{0,1\}$, where:

- $\pi_1 \circ \mathsf{s}(q)\colon a \mapsto \delta(q,a)$, for arbitrary state $q \in Q$ and letter $a \in \Sigma$.
- $\pi_2 \circ \mathsf{s}(q) = 1$ if and only if $q \in F$.

That is to say, the two functors $(\cdot)^\Sigma$ and $\{0,1\}$ captures exactly the δ and F parts of the automata, respectively.

More importantly, the final $(\cdot)^\Sigma \times \{0,1\}$-coalgebra exists, say $\zeta\colon \mathcal{P}\Sigma^* \to (\mathcal{P}\Sigma^*)^\Sigma \times \{0,1\}$, where Σ^* is the set of all finite strings over Σ. In particular, given a set of strings $\Gamma \subseteq \Sigma^*$, $\pi_1 \circ \zeta(\Gamma)\colon (a \in \Sigma) \mapsto \{\tau \in \Sigma^* \mid a\tau \in \Gamma\}$, and $\pi_2 \circ \zeta(\Gamma) = 1$ precisely if $\epsilon \in \Gamma$ The finality of ζ induces a unique coalgebra morphism $!_{\mathsf{s}}\colon \mathsf{s} \to \mathcal{P}\Sigma^*$, which maps a state q to exactly the set of strings accepted at q.

2.2.3 Symmetric monoidal categories

We recall the basics of symmetric monoidal categories.

Definition 2.2.10. A *monoidal category* $\langle \mathbb{C}, \otimes, I\rangle$ is $\mathbb{C}$ equipped with:

- a bifunctor $\otimes\colon \mathbb{C} \times \mathbb{C} \to \mathbb{C}$ called the *monoidal* or *tensor product*, whose associativity is witnessed by a natural isomorphism $\alpha_{A,B,C}\colon A \otimes (B \otimes C) \to (A \otimes B) \otimes C$

- an object I called the *monoidal unit* with natural isomorphisms $\lambda_A \colon I \otimes A \to A$ and $\rho_A \colon A \otimes I \to A$,

Moreover, they commute the following diagrams representing coherence of associativity and unitality:

$$\begin{tikzcd}
A \otimes (B \otimes (C \otimes D)) \arrow[r, "\alpha_{A,B,C\otimes D}"] \arrow[d, "\alpha_{A,B,C\otimes D}"'] & (A \otimes B) \otimes (C \otimes D) \arrow[r, "\alpha_{A\otimes B,C,D}"] & ((A \otimes B) \otimes C) \otimes D \arrow[d, "\alpha_{A,B,C}\otimes id_D"] \\
A \otimes ((B \otimes C) \otimes D) \arrow[rr, "\alpha_{A,B\otimes C,D}"] & & (A \otimes (B \otimes C)) \otimes D
\end{tikzcd}$$

$$\begin{tikzcd}
A \otimes (I \otimes B) \arrow[rr, "\alpha_{A,I,B}"] \arrow[dr, "id_A\otimes\lambda_B"'] & & (A \otimes I) \otimes B \arrow[dl, "\rho_A\otimes id_B"] \\
& A \otimes B &
\end{tikzcd}$$

A monoidal category $\mathbb{C}$ is *symmetric* if there is a class of natural isomorphisms $\sigma_{A,B} \colon A \otimes B \to B \otimes A$ for any objects A, B, that is:

- self-inverse: $\sigma_{B,A} \circ \sigma_{A,B} = id_{A\otimes B}$,
- compatible with I:

$$\begin{tikzcd}
A \otimes I \arrow[rr, "\sigma_{A,I}"] \arrow[dr, "\rho_A"'] & & I \otimes A \arrow[dl, "\lambda_A"] \\
& A &
\end{tikzcd}$$

- compatible with α:

$$\begin{tikzcd}
(A \otimes B) \otimes C \arrow[r, "\sigma_{A,B}\otimes id_C"] \arrow[d, "\alpha_{A,B,C}"'] & (B \otimes A) \otimes C \arrow[r, "\alpha_{B,A,C}"] & B \otimes (A \otimes C) \arrow[d, "id_B\otimes\sigma_{A,C}"] \\
A \otimes (B \otimes C) \arrow[r, "\sigma_{A,B\otimes C}"] & (B \otimes C) \otimes A \arrow[r, "\alpha_{B,C,A}"] & B \otimes (C \otimes A)
\end{tikzcd}$$

We say a monoidal category $\mathbb{C}$ is *strict* if α, λ and ρ are all identities.

Definition 2.2.11. Let $(\mathbb{C}, \otimes, I_{\mathbb{C}})$ and $(\mathbb{D}, \oplus, I_{\mathbb{D}})$ be two monoidal categories. A *monoidal functor* $\mathcal{F} \colon \mathbb{C} \to \mathbb{D}$ is a functor together with

- a natural transformation $\phi_{A,B} \colon \mathcal{F}A \oplus \mathcal{F}B \to \mathcal{F}(A \otimes B)$,
- a morphism $\phi_I \colon I_{\mathbb{D}} \to \mathcal{F}I_{\mathbb{C}}$

satisfying the following conherence conditions which state the compatibility with α, λ and ρ:

$$\begin{tikzcd}
(\mathcal{F}A \oplus \mathcal{F}B) \oplus \mathcal{F}C \arrow[r, "\alpha^{\mathbb{D}}_{\mathcal{F}A,\mathcal{F}B,\mathcal{F}C}"] \arrow[d, "\phi_{A,B}\oplus id_{\mathcal{F}C}"'] & \mathcal{F}A \oplus (\mathcal{F}B \oplus \mathcal{F}C) \arrow[r, "id_{\mathcal{F}A}\oplus\phi_{B,C}"] & \mathcal{F}A \oplus \mathcal{F}(B \otimes C) \arrow[d, "\phi_{A,B\otimes C}"] \\
\mathcal{F}(A \otimes B) \oplus \mathcal{F}C \arrow[r, "\phi_{A\otimes B,C}"] & \mathcal{F}((A \otimes B) \otimes C) \arrow[r, "\mathcal{F}\alpha^{\mathbb{C}}_{A,B,C}"] & \mathcal{F}(A \otimes (B \otimes C))
\end{tikzcd}$$

$$\begin{tikzcd}
\mathcal{F}A \oplus I_{\mathbb{D}} \arrow[r, "id_{\mathcal{F}A} \oplus \phi_I"] \arrow[d, "\rho^{\mathbb{D}}_{\mathcal{F}A}"'] & \mathcal{F}A \oplus \mathcal{F}I_{\mathbb{C}} \arrow[d, "\phi_{A,I_{\mathbb{C}}}"] \\
\mathcal{F}A & \mathcal{F}(A \otimes I_{\mathbb{C}}) \arrow[l, "\mathcal{F}\rho^{\mathbb{C}}_A"]
\end{tikzcd}
\qquad
\begin{tikzcd}
I_{\mathbb{C}} \oplus \mathcal{F}B \arrow[r, "\phi_I \otimes id_{\mathcal{F}A}"] \arrow[d, "\lambda^{\mathbb{D}}_{\mathcal{F}B}"'] & \mathcal{F}I_{\mathbb{C}} \oplus \mathcal{F}B \arrow[d, "\phi_{I_{\mathbb{C}},B}"] \\
\mathcal{F}B & \mathcal{F}(I_{\mathbb{C}} \otimes B) \arrow[l, "\mathcal{F}\lambda^{\mathbb{C}}_B"]
\end{tikzcd}$$

In Definition 2.2.11, if both $\phi_{A,B}$ and ϕ_I are isomorphisms, then $\mathcal{F}$ is *strong monoidal*; if they are identities, then they are *strict monoidal.*

Definition 2.2.12. Given two monoidal functors $(\mathcal{F}, \phi)$ and $(\mathcal{G}, \psi)$ of type $\mathbb{C} \to \mathbb{D}$, a *monoidal natural transformation* between $\mathcal{F}$ and $\mathcal{G}$ is a natural transformation $\theta \colon \mathcal{F} \Rightarrow \mathcal{G}$ that is compatible with ϕ and ψ:

$$\begin{tikzcd}
\mathcal{F}A \oplus \mathcal{F}B \arrow[r, "\phi_{A,B}"] \arrow[d, "\theta_A \oplus \theta_B"'] & \mathcal{F}(A \otimes B) \arrow[d, "\theta_{A \otimes B}"] \\
\mathcal{G}A \oplus \mathcal{G}B \arrow[r, "\psi_{A,B}"] & \mathcal{G}(A \otimes B)
\end{tikzcd}
\qquad
\begin{tikzcd}
& I_{\mathbb{D}} \arrow[dl, "\phi_I"'] \arrow[dr, "\psi_I"] & \\
\mathcal{F}I_{\mathbb{C}} \arrow[rr, "\theta_{I_{\mathbb{C}}}"] & & \mathcal{G}I_{\mathbb{C}}
\end{tikzcd}$$

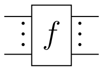
The following notion of equivalence is useful for the statement of coherence theorem (**??**). Put it simple, it is just the monoidal variant of equivalence between categories.

Definition 2.2.13. Two monoidal categories $\mathbb{C}$ and $\mathbb{D}$ are *monoidal equivalent* if there exist two strong monoidal functors $\mathcal{F} \colon \mathbb{C} \to \mathbb{D}$ and $\mathcal{G} \colon \mathbb{D} \to \mathbb{C}$ equipped with monoidal natural transformations $\eta \colon \mathbf{1}_{\mathbb{C}} \Rightarrow \mathcal{G} \circ \mathcal{F}$ and $\varepsilon \colon \mathcal{F} \circ \mathcal{G} \Rightarrow \mathbf{1}_{\mathbb{D}}$, where $\mathbf{1}$ is the identity functor on the given category.

2.3 Graphical calculus

We recall the notion of string diagrams and diagrammatic presentation of monoidal categories.

2.3.1 Categories of string diagrams

Intuitively, one can construct a monoidal category 'freely' using some diagrammatic components of the form  as 'lego blocks'. Such diagrams are naturally equipped with the structure of monoidal categories: the sequential composition of two such pieces is simply connecting the outcome wires of one block with the input wires of another, and the monoidal product is juxtapositoin.

In this subsection we illustrate the idea and the bpower of diagrammatic calculus using the graphical linear algebra (GLA) from Bonchi et al. [BSZ17]. Note that while the construction there works for arbitrary fields $\mathbb{K}$, we adopt a toy version here by instantiating the theory to the field $\langle \mathbb{Z}, +, \cdot, 0, 1 \rangle$ of integers.

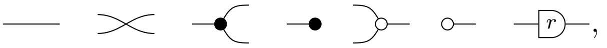
We start with a diagrammatic calculus for matrices over $\mathbb{Z}$.

Definition 2.3.1. In the category $\mathsf{Mat}\,\mathbb{Z}$ of matrices over integers, objects are natural numbers, and morphisms $m \to n$ are $n \times m$-matrices in $\mathbb{Z}$. $\mathsf{Mat}\,\mathbb{Z}$ together with the direct sum on matrices form a (symmetric) monoidal category.

The diagrammatic calculus $\mathbb{HA}_{\mathbb{Z}}$ for $\mathbb{Z}$-matrices is a category of string diagrams inductively defined (or, freely generated) as follows.

Definition 2.3.2. The symmetric monoidal category Syn has the following components:

- the objects are natural numbers;
- the morphisms are string diagrams freely composed using the following ingredients:



, $r \in \mathbb{Z}$

such that morphisms $m \to n$ are diagrams with m wires on the left port, and n wires on the right port.

In particular, parallel and sequential compositions are the juxtaposition and sequential composition of diagrams. The category $\mathbb{HA}_{\mathbb{Z}}$ of *Hopf algebra over* $\mathbb{Z}$ is obtained by quotienting the morphisms of Syn by the axioms in Figure 2.1.

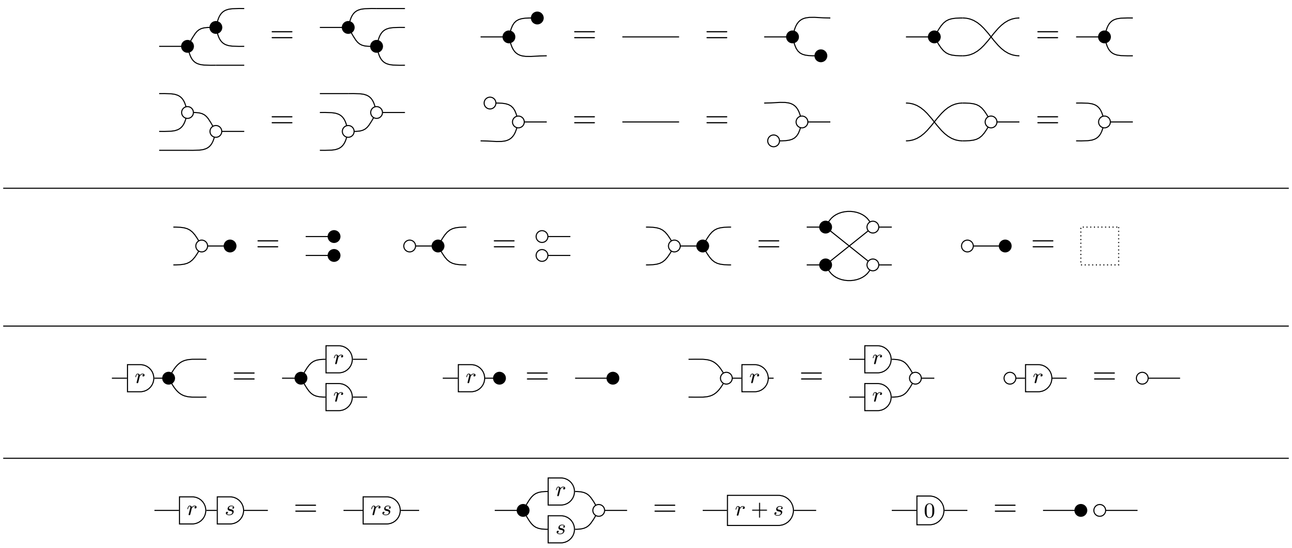
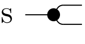
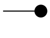
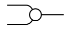
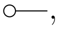
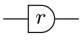
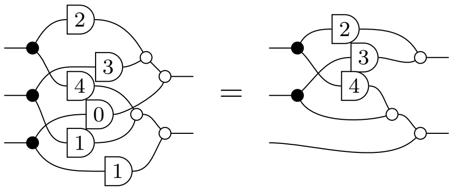
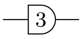


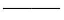
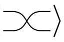
Figure 2.1: Hopf Algebra axioms for $\mathbb{HA}_{\mathbb{Z}}$

Indeed both Syn and $\mathbb{HA}_{\mathbb{Z}}$ are strict symmetric monoidal categories. We say Syn is the SMC freely generated with the generating object 1 and generating morphisms , , , ,  for all integers r; $\mathbb{HA}_{\mathbb{Z}}$ is the freely generated SMC with the same generators and moreoever *quotiented by* the equational theory in Fig. 2.1. In fact $\mathbb{HA}_{\mathbb{Z}}$ admits a finite set of generators and a finite axiomatisation, which we refer to [BSZ17] for details.

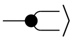
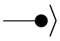
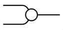
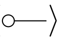
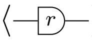
Even before giving a precise semantics of the diagrams, there is a intuitive reading of matrices as diagram. For example, the matrix $A = \begin{pmatrix} 2 & 3 & 0 \\ 4 & 0 & 1 \end{pmatrix}$ is diagrammatically expressed as



For instance, the entry $A_{12} = 3$ corresponds to the diagram where the second wire on the left port has one copy that meets a scalar  and is added up to the first wire on the right port. This connection is formalised as a functor $\langle - \rangle : \mathbb{HA}_{\mathbb{Z}} \to \mathsf{Mat}\ \mathbb{Z}$ that interprets every diagram as a matrix. Given the inductive nature of $\mathbb{HA}_{\mathbb{Z}}$, the interpretation can be defined inductively. The following definition is implicit — for instance $\langle - \rangle$ maps the monoidal structure to the monoidal structure.

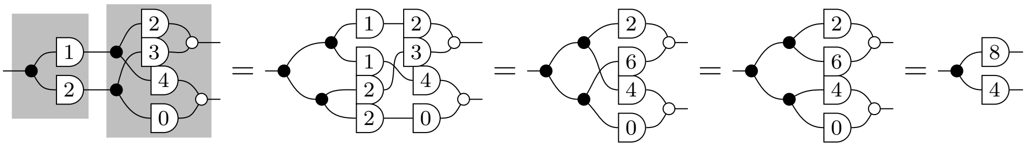
$\langle$  $\rangle = 1$ $\quad \langle$  $\rangle = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \quad \langle c \,;\, d \rangle = \langle d \rangle \cdot \langle c \rangle \quad \langle c \otimes d \rangle = \begin{pmatrix} \langle c \rangle & \mathbf{0} \\ \mathbf{0} & \langle d \rangle \end{pmatrix}$

For the generating morphisms specific for GLA, their semantics are defined as follows:

$\langle$  $\rangle = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$ $\quad \langle$  $\rangle = ¡$ $\quad \langle$  $\rangle = \begin{pmatrix} 1 & 1 \end{pmatrix}$ $\quad \langle$  $\rangle = \, !$ $\quad \langle$  $\rangle = r,$ for all $r \in \mathbb{Z}$

where $¡ \colon 1 \to 0$ and $! \colon 0 \to 1$ are the unique morphisms given by the universal property of 0 (as both the final and initial object) in $\mathsf{Mat}\ \mathbb{Z}$.

While for a single matrix things look straightforward, it is no longer the case when operations on matrices come into the picture. In particular, the axioms for $\mathbb{HA}_{\mathbb{Z}}$ provides enough power to calculate the direct sum and multiplication of matrices diagrammatically. For instance, the diagrammatic counterpart of $\begin{pmatrix} 2 & 3 \\ 4 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ 2 \end{pmatrix} = \begin{pmatrix} 8 \\ 4 \end{pmatrix}$ is as follows:



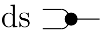
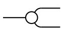
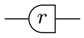
That $\mathbb{HA}_{\mathbb{Z}}$ is exactly the diagrammatic counterpart of $\mathsf{Mat}\ \mathbb{Z}$ with direct sum as monoidal product is formally the following proposition, in which case we say $\mathbb{HA}_{\mathbb{Z}}$ *presents* $\mathsf{Mat}\ \mathbb{Z}$.

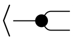
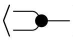
Proposition 2.3.3. $\mathbb{HA}_{\mathbb{Z}} \cong \mathsf{Mat}\ \mathbb{Z}$

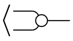
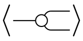
In particular, one direction of the isomorphism is witnessed by the interpretation functor $\langle - \rangle$.

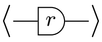
One important conceptual switch that is important for Section 5.1.2 is to turn from a *functional* semantics to a *relational* semantics. In the case of GLA, this switch is from the category of matrices in $\mathbb{Z}$ to the category of linear relations over $\mathbb{Q}$.

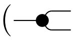
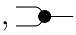
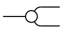
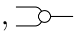
Definition 2.3.4. A *linear relation* $R\colon m \to n$ over $\mathbb{Q}$ is a linear subspace of $\mathbb{Q}^m \times \mathbb{Q}^n$. The symmetric monoidal category $\mathsf{LinRel}_{\mathbb{Q}}$ of linear relations has rational numbers as objects, and linear relations over $\mathbb{Q}$ as morphisms. Sequential and parallel compositions are relation composition and product, respectively.

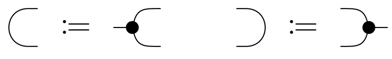
The diagrammatic calculus for $\mathsf{LinRel}_{\mathbb{Q}}$ is then an enrichment of $\mathbb{HA}_{\mathbb{Z}}$ which adds  and  (together with their units) as well as , whose intuition is to flow information from right to left. To match the relational semantics, diagrams are interpreted as relations instead of functions via some functor $\langle - \rangle : \mathbb{HA}_{\mathbb{Z}} \to \mathsf{LinRel}_{\mathbb{Q}}$. For instance,

$\langle$  $\rangle = \{(x,(x,x)) \mid x \in \mathbb{Q}\}$ $\qquad$ $\langle$  $\rangle = \{((y,y),y) \mid y \in \mathbb{Q}\}$

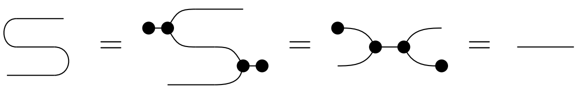
$\langle$  $\rangle = \{((x_1,x_2),y) \in \mathbb{Q}^2 \times \mathbb{Q} \mid x_1 + x_2 = y\}$ $\qquad$ $\langle$  $\rangle = \{(x,(y_1,y_2)) \in \mathbb{Q} \times \mathbb{Q}^2 \mid x = y_1 + y_2\}$

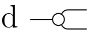
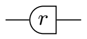
$\langle$  $\rangle = \{(x,y) \in \mathbb{Q} \times \mathbb{Q} \mid r \cdot x = y\}$

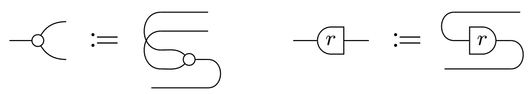
The axioms for $\mathbb{IH}_{\mathbb{Z}}$ are shown in Figure 2.2. In particular, the third block states that ( , ) and ( , ) are two *Frobenius algebra*, such that only the connectivity matters. One important consequence is that the Frobenius axioms induce a (self-dual) compact closed structure where 'cups' and 'caps' interpret feedback:



such that bended wires (using and) can be pulled straight:

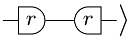
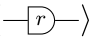
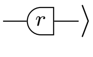


Then we regard  and  as abbreviations:



Theorem 2.3.5 ([BSZ17])**.** $\mathbb{IH}_{\mathbb{Z}} \cong \mathsf{LinRel}_{\mathbb{Q}}$

It is helpful to check that some of the axioms semantically hold (soundness). For instance,

$\langle$  $\rangle = \langle$  $\rangle \mathbin{;} \langle$  $\rangle$

$$= \{(x,y) \in \mathbb{Q} \times \mathbb{Q} \mid rx = y\} \mathbin{;} \{(u,v) \in \mathbb{Q} \times \mathbb{Q} \mid u = rv\}$$
$$= \{(x,v) \in \mathbb{Q} \times \mathbb{Q} \mid \exists u \in \mathbb{Q}\colon rx = u \text{ and } u = rv\}$$

$$
\begin{aligned}
&= \{(x,v) \in \mathbb{Q}\times\mathbb{Q} \mid rx = rv\}\\
&= \{(x,v) \in \mathbb{Q}\times\mathbb{Q} \mid x = v\} = \langle\text{——}\rangle
\end{aligned}
$$

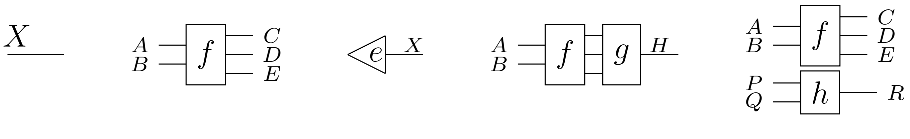
$$
\begin{aligned}
\langle -\boxed{r}-\boxed{r}- \rangle &= \langle -\boxed{r}- \rangle \mathbin{;} \langle -\boxed{r}- \rangle\\
&= \{(x,y) \in \mathbb{Q}\times\mathbb{Q} \mid x = ry\} \mathbin{;} \{(u,v) \in \mathbb{Q}\times\mathbb{Q} \mid ru = v\}\\
&= \{(x,v) \mid \exists y \in \mathbb{Q}\colon x = ry = v\}\\
&= \langle\text{——}\rangle
\end{aligned}
$$

Comparing the two calculation, we notice that while $\langle -\boxed{r}-\boxed{r}- \rangle = \langle\text{——}\rangle$ holds regardless of the semantic domain being $\mathbb{Z}$ or $\mathbb{Q}$, the last step showing $\langle -\boxed{r}-\boxed{r}- \rangle = \langle\text{——}\rangle$ relies on the fact that variables take values in rationals rather than integers: let $r = 2$, then those integers satisfying that $\exists y \in \mathbb{Z}$ such that $x = 2y = v$ are only even integers.

The above diagrammatic representation — drawing objects as wires and morphisms as boxes — works not only for strict monoidal categories, but also for *non-strict* monoidal categories in general. This is a consequence of the following result from Mac Lane [Mac71] (note that the terminology 'Coherence Theorem' refers to several statements, and we only mention one of them here).

Theorem 2.3.6 (Coherence Theorem [Mac71])**.** *Every monoidal category is monoidally equivalent to a strict monoidal category.*

As a consequence, we can unambiguously write $A\otimes B\otimes C$ and $f\otimes g\otimes h$ without specifying the bracketing until necessary — the Coherence Theorem guarantees that one can unambiguously add the brackets whenever necessary. The diagram below are id_X, $f\colon A\otimes B \to C\otimes D\otimes E$, $e\colon I \to X$, and two instances of sequential and parallel composition of morphisms:



While morphisms are in general drawn as boxes, we use triangles when the (co)domain is known to be the tensor unit I. We emphasise that such representation is mathematically rigorous, see for example Joyal and Street [JS91] for a soundness and completeness result for the diagrammatic axiomatisation for monoidal categories. Selinger [Sel10] is a survey of the diagrammatic representation of various monoidal categories.

We illustrate the power and flexibility of the diagrammatic presentation of monoidal categories via some examples.

We start with symmetric monoidal categories (see Definition 2.2.10). The natural isomorphism $\sigma_{A,B}$ is drawn as $^{A}_{B}\!\times\!^{B}_{A}$, and the three conditions for the symmetric structure amounts

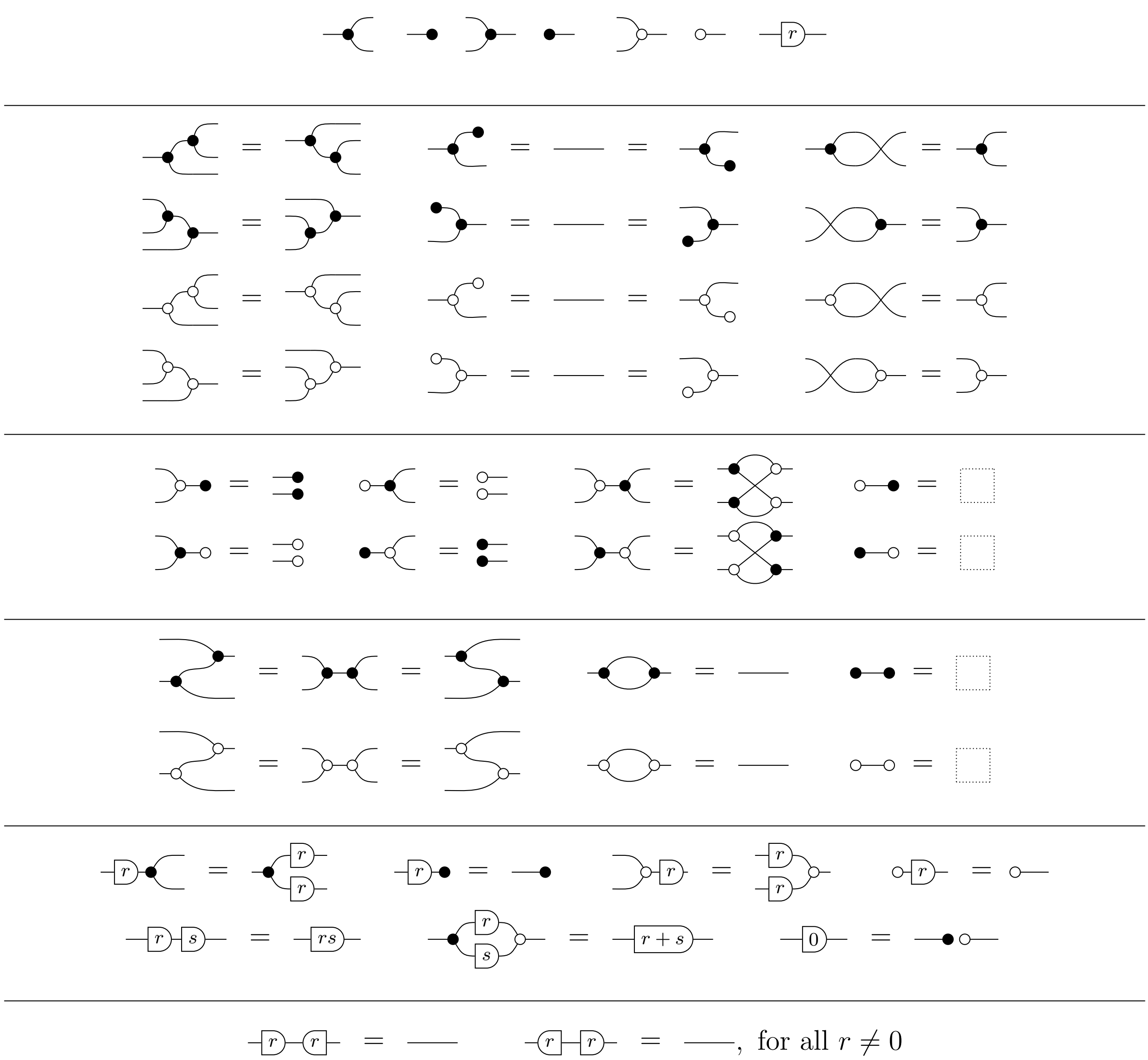
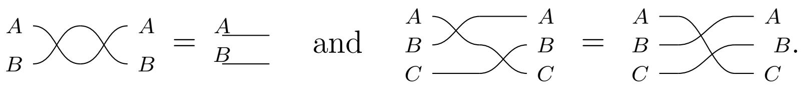


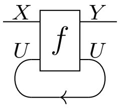
Figure 2.2: Interacting Hopf Algebra $\mathbb{IH}_{\mathbb{Z}}$

to



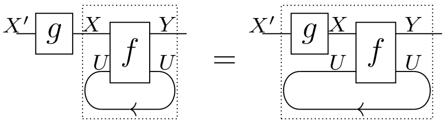
The next example is *traced monoidal categories*, which will also be used in Subsection **??**. While first introduced in the context of balanced monoidal categories by Joyal *et al.* [JSV96], we adopt the definition from Hasegawa [Has97] stated for symmetric monoidal categories. In Definition 2.3.7 below, one can compare the diagrammatic and traditional statement of the conditions to see how diagrams simplify and visualise the reasoning.

Definition 2.3.7 (Traced monoidal categories)**.** A symmetric monoidal category $\langle \mathbb{C}, \otimes, I \rangle$ is *traced* if there is a family of morphisms $Tr^{U}_{X,Y} \colon \mathbb{C}(X \otimes U, Y \otimes U) \to \mathbb{C}(X, Y)$ for each objects X, Y, U, such that for arbitrary $f \colon X \otimes U \to Y \otimes U$, $Tr^{U}_{X,Y}$ is depicted diagrammatically as

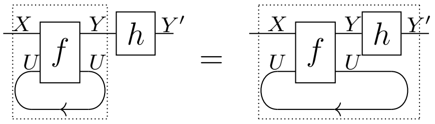


, and satisfies the following conditions:

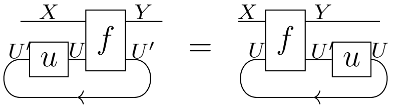
1. Naturality in X: for arbitrary $f\colon X \otimes U \to Y \otimes U$ and $g\colon X' \to X$, $Tr^U_{X,Y}(f) \circ g = Tr^U_{X',Y}(f \circ (g \otimes id_U))$.



2. Naturality in Y: for arbitrary $f\colon X \otimes U \to Y \otimes U$ and $h\colon Y \to Y'$, $h \circ Tr^U_{X,Y}(f) = Tr^U_{X,Y'}((h \otimes id_U) \circ f)$.

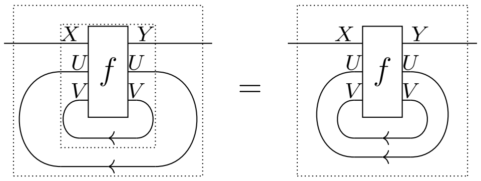


3. Dinaturality in U: for arbitrary $f\colon X \otimes U \to Y \otimes U'$ and $u\colon U' \to U$, $Tr^{U'}_{X,Y}(f \circ (id_X \otimes u)) = Tr^U_{X,Y}((id_Y \otimes u) \circ f)$.

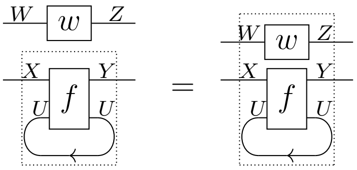


4. Vanishing (i): for arbitrary $f\colon X \otimes I \to Y \otimes I$, $Tr^I_{X,Y}(f) = \rho_Y^{-1} \circ f \circ \lambda_X$.

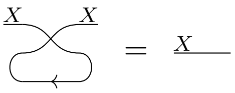
5. Vanishing (ii): for arbitrary $f\colon X \otimes U \otimes V \to Y \otimes U \otimes V$, $Tr^U_{X,Y}(Tr^V_{X\otimes U, Y\otimes U}(f)) = Tr^{U\otimes V}_{X,Y}(f)$.



6. Superposing: for arbitrary $f\colon X \otimes U \to Y \otimes V$ and $w\colon W \to Z$, $w \otimes Tr^U_{X,Y}(f) = Tr^U_{W\otimes X, Z\otimes Y}(w \otimes f)$.



7. Yanking: for arbitrary object X, $Tr^X_{X,X}(\sigma_X) = id_X$.



To our purpose, it is useful to think of the traces as feedback.

Example 2.3.8. ([Has97]) The category $\mathsf{CPO}^{\perp}$ of pointed cpo's and continuous functions is a traced monoidal category. Recall that 'cpo' stands for *complete partial order*, namely partial orders in which ω-indexed chains always have suprema. It is *pointed* if it is equipped with a bottom element $\bot$. A function $f\colon X \to Y$ between two pointed cpo is *continuous* if it preserves the suprema of any ω-indexed chain. The symmetric monoidal structure is products of posets and functions, which indeed is cartesian.

Given a $\mathsf{CPO}^{\bot}$-morphism $f\colon X \times U \to Y \times U$, to define the trace $Tr^{U}_{X,Y}\colon X \to Y$, let $u_x := \mu u.\pi_2 \circ f(x,u)$, then

$$Tr^{U}_{X,Y}(x) = f(x,u_x) \tag{2.4}$$

In other words, u_x is the least u satisfying that $f(x,u) = (y,u)$ for some y, whose existence is guaranteed by the fact that f is continuous on the pointed cpos.

We could verify the naturality in Y as an instance. Consider $f\colon X \times U \to Y \times U$ and $h\colon U \to U'$. We show that, for an arbitrary $x \in X$, $h \circ Tr^{U,X,Y}_{f}(x) = Tr^{U}_{X,Y'}((h \times id_U) \circ f)(x)$. The following least fixed points are necessary

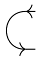
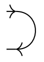
$$\begin{aligned} u_x &:= \mu u.\pi_2 \circ f(x,u) \\ v_x &:= \mu v.\pi_2 \circ ((h \times id_U) \circ f(x,v)) \end{aligned}$$

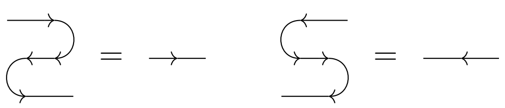
Thus $y_1 := h \circ Tr^{U,X,Y}_{f}(x) = h(\pi_1 \circ f(x,u_x))$, and $y_2 := Tr^{U}_{X,Y'}((h \times id_U) \circ f)(x) = (\pi_1 \circ (h \times id_U) \circ f)(x,v_x)$. It suffices to show that $u_x = v_x$, since this implies $y_1 = y_2$ as follows

$$\begin{aligned} (y_2, v_x) &= ((h \times id_U) \circ f)(x,v_x) \\ &= ((h \times id_U) \circ f)(x,u_x) \\ &= (h \times id_U)(f(x,u_x)) \\ &= (h \times id_U)(\pi_1 \circ f(x,u_x), \pi_2 \circ f(x,u_x)) \\ &= (h \circ \pi_1 \circ f(x,u_x), id_U \circ \pi_2 \circ f(x,u_x)) \\ &= (y_1, u_x) \end{aligned}$$

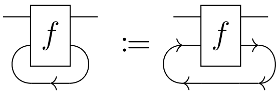
Now we prove $u_x = v_x$ by the two inequalities. On one hand, $\pi_2 \circ (h \times id_U) \circ f(x,u_x) = \pi_2 \circ (h \circ \pi_1 \circ f(x,u_x), \pi_2 \circ f(x,u_x)) = \pi_2 \circ f(x,u_x) = u_x$, thus $v_x \leq u_x$. On the other hand, $\pi_2 \circ f(x,v_x) = \pi_2 \circ ((h \times id_U) \circ f(x,v_x)) = v_x$, thus $u_x \leq v_x$.

The traced structure in $\mathsf{CPO}^{\bot}$ is used to interpret while loops in imperative programming languages, as certain least fixed points. The details can be found in, for example, Mitchell [Mit96].

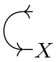
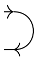
A closely related concept is (self-dual) compact closed categories, which we refer to Kelly and Laplaza [KL80] for details and the diagrammatic presentation. In short, every object (in diagrammatic format) $\xrightarrow{X}$ has a dual $\xleftarrow{X}$ such that there are 'cup'  and 'cap'  morphisms satisfying the following snake equation:



Every compact closed category has canonical traces formed of the cups and caps, namely



Interestingly, for the other direction, every compact closed category arises from a traced moniodal category using so-called *Int* construction (standing for 'integers' as the construction 'categorifies' that of integers from natural numbers). Detailed discussion on the connection between traced and compact closed categories can be found in Joyal *et al.* [JSV96].

Example 2.3.9. The category Rel of relations is one of the simplest examples of compact closed categories. In short, Rel has sets as objects, and relations as morphisms denoted as $A \nrightarrow B$. Composition of morphisms is relational composition: given relations $R_1 \colon X \nrightarrow Y$ and $R_2 \colon Y \fatsemi Z$, their relational composition $R_1 \fatsemi R_2 \colon X \nrightarrow Z$ is defined as $\{(x,z) \in X \times Z \mid \exists y \in Y : (x,y) \in R_1 \text{ and } (y,z) \in R_2\}$. Category Rel with relation products and the canonical singleton set $\mathbf{1} = \{\bullet\}$ form a SMC, where the relation product is $R_1 \times R_3 \colon X \times U \nrightarrow Y \times V$ is defined as $\{((x,u),(y,v)) \in (X \times U) \times (Y \times V) \mid (x,y) \in R_1 \text{ and } (u,v) \in R_3\}$. Moreover, Rel is compact closed: every object X is self-dual, with  $_X = \{(\bullet,(x,x)) \mid x \in X\}$ and  $_X = \{((x,x),\bullet) \mid x \in X\}$.

Consequently, $\langle \mathsf{Rel}, \times, \mathbf{1}\rangle$ is traced, where for an arbitrary relation $R \colon X \times U \nrightarrow Y \times U$, $Tr^U_{X,Y}(R) = \{(x,y) \in X \times Y \mid \exists u \in U \text{ such that } ((x,u),(y,u)) \in R\}$.

As another application of diagrammatic calculus for monoidal categories, we recall the notion of CD categories.

Definition 2.3.10. A *copy-discard category*, or *CD category* in short, is a SMC $\langle \mathbb{C}, \otimes, I\rangle$ equipped with two classes of morphisms called *duplicators* $\nabla_X \colon X \to X \otimes X$ and *dischargers* $\epsilon_X \colon X \to I$ satisfying the following conditions:

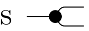
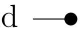
1. $(\nabla_X \otimes id_X) \circ \nabla_X = (id_X \otimes \nabla_X) \circ \nabla_X$.

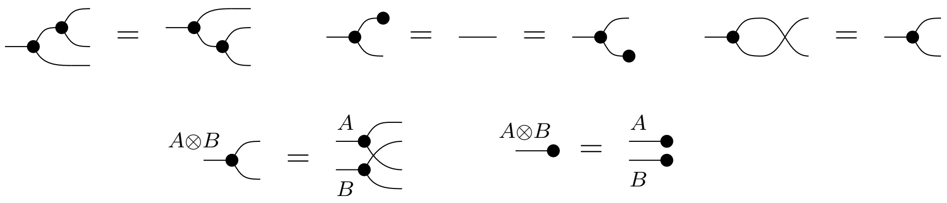
2. $(\epsilon_X \otimes id_X) \circ \nabla_X = (id_X \otimes \epsilon_X) \circ \nabla_X = id_X$.

3. $\sigma_{X,Y} \circ \nabla_X = \nabla_X$.

4. $\nabla_{X\otimes Y} = (id_X \otimes \sigma_{X,Y} \otimes id_Y) \circ (\nabla_X \otimes \nabla_Y)$.

5. $\epsilon_{X\otimes Y} = \epsilon_X \otimes \epsilon_Y$.

The intuition of ∇ and ϵ is — as their names suggest — to duplicate an object or to discard it. In Definition 2.3.10, the conditions can be read as: ∇_X is associative; ϵ_X is the unit of ∇_X; ∇_X is compatible with σ_X; ∇_X and ϵ_X are both compatible with $\otimes$. Using a diagrammatic representation of ∇ and ϵ as  and , the defining properties of CD categories can be expressed as follows:



(2.5)

Note that every Cartesian category is automatically a CD category, where the duplicators and dischargers are the diagonal morphisms and unique morphisms to the terminal object. Indeed, a CD category is cartesian if and only if all the duplicators and dischargers are *natural* [Fox76]: for arbitrary morphism $f\colon X \to Y$, $\nabla_Y \circ f = (f \otimes f) \circ \nabla_X$, $\epsilon_Y \circ f = \epsilon_X$. We can equip CD categories with the 'duals' of ∇ and ϵ, say $\Delta_X\colon X \times X \to X$ and $\eta\colon I \to X$, respectively, satisfying the dual of the conditions in Definition 2.3.10 (or in other words, satisfying the equations in (2.5) 'flipped along the y-axis'). We refer to such categories as CDMU categories (for **c**opy, **d**iscard, **m**ultiplication, **u**nit).

Chapter 3

Coalgebraic semantics for logic programming

In this chapter we explore PLP and WLP from a coalgebraic point of view, in the aim of capturing the goal-directed behaviour of the programs. Recall that in the case of LP, such goal-directed behaviour amounts to reducing the task of proving one goal to the subtasks of proving multiple subgoals, according to available clauses from the program. The situation is similar in the case of PLP and WLP, only that one replaces 'proving a goal' with 'calculating the probability/weight of a goal'. In both cases, we define certain notion of trees that capture such (non-deterministic) goal-directed behaviour of PLP and WLP programs, from which one can retrieve the standard semantics.

We provide such a coalgebraic semantics for both the ground case (i.e. without variable) and the general case (i.e. possibly with variables). In the ground case, the atoms carry a simple structure of sets, and this relatively simple case serves to clarify our main coalgebraic constructs. In particular, both the coalgebraic presentation of PLP and WLP programs and their coalgebraic semantics are already defined at this stage. Our presentation is inspired by the coalgebraic treatment of LP in which ground programs are in 1-1 correspondence with coalgebras for the functor $\mathcal{P}_f\mathcal{P}_f$, and the coalgebraic semantics is defined using the unviersal property of the final coalgebra.

The general case adds upon the ground case extra categorical structure to deal with variables and substitutions. Instead of sets, the space of atoms are now a presheaf indexed by a Lawvere theory of terms and substitutions, a standard categorical tool to deal with term algebra. We shall follow the saturation approach in Bonchi and Zanasi [BZ15] to define the coalgebraic semantics which encodes unification.

In light of the coalgebraic treatment of pure logic programming [KMP10, KP11, BZ13, BZ15], the generalisation to PLP and WLP may not appear so surprising. We believe the importance is two-fold. First, whereas the derivation semantics $[\![-]\!]$ (Definition 3.1.23) is a straight generalisation of the pure setting, the distribution semantics $\langle\!\langle - \rangle\!\rangle$ (Definition 3.1.33) is genuinely novel , and does not have counterpart in pure logic programming. Second, this work provides a starting point for a generic coalgebraic treatment of variations of logic programming:

- The coalgebraic approach to pure logic programming has been used as a formal justification [KPS16, KL17] for coinductive logic programming [KL18, GBM$^+$07]. Coinduction in the context of probabilistic and weighted logic programs is, to the best of our knowledge, a completely unexplored field, for which the current paper establishes semantic foundations.

- As mentioned, reasoning in Bayesian networks can be seen as a particular case of PLP, equipped with the distribution semantics. Our coalgebraic perspective thus readily applies to Bayesian reasoning, paving the way for combination with recent works [JKZ19, JZ19, DDGK16] modelling belief revision, causal inference and other Bayesian tasks in algebraic terms.

3.1 Probabilistic logic programming: propositional case

This section is devoted to a coalgebraic perspective of propositional PLP. We first introduce two coalgebraic semantics of PLP, one corresponds to a reductive reading of (probabilistic) logic programs, and the other relates closely to the distribution semantics of PLP programs. Then we present how to recover the distribution semantics from the coalgebraic semantics via an algorithm.

We follow the canonical procedure in Section 2 to define coalgebraic semantics for transition systems. In our case, our transition systems are propositional PLP programs, and the first step is to find a coalgebraic treatment of them. Inspired by the coalgebraic treatment of 'pure' logic programming from Komendantskaya et al. [KMP10], we observe that propositional PLP programs are in 1-1 correspondence with coalgebras for the functor $\mathcal{M}_{pr}\mathcal{P}_f$, where $\mathcal{M}_{pr}$ is the probabilistic multiset functor on $[0, 1]$ and $\mathcal{P}_f$ is the finite powerset functor. We then provide two coalgebraic semantics for propositional PLP.

- The first interpretation $[\![-]\!]$ is in terms of execution trees called *stochastic derivation trees*, which represent parallel SLD-derivations of a program on a goal. Stochastic derivation trees are the elements of the cofree $\mathcal{M}_{pr}\mathcal{P}_f$-coalgebra on a given set of atoms At, meaning that any goal $a \in At$ can be given a semantics in terms of the corresponding stochastic derivation tree by the universal map $[\![-]\!]$ to the cofree coalgebra.

- The second interpretation $\langle\!\langle - \rangle\!\rangle$ recovers the usual distribution semantics of PLP. We introduce *distribution trees*, a tree-like representation of the distribution semantics, as the elements of the cofree $\mathcal{D}_{\leq}\mathcal{P}_f\mathcal{P}_f$-coalgebra on At, where $\mathcal{D}_{\leq}$ is the sub-probability distribution monad (see Example 2.2.7). In order to characterise $\langle\!\langle - \rangle\!\rangle$ as the map given by universal property of distribution trees, we need a canonical extension of PLP to the setting of $\mathcal{D}_{\leq}\mathcal{P}_f\mathcal{P}_f$-coalgebras. This is achieved via a 'possible worlds' natural transformation $\mathcal{M}_{pr} \Rightarrow \mathcal{D}_{\leq}\mathcal{P}_f$.

In the remainder of this section, we fix a finite set of atoms $At = \{A_1, \ldots, A_n\}$ and a PLP program $\mathbb{P}$ based on At, unless specified otherwise.

3.1.1 Representing propositional PLP as coalgebras

In aim of obtaining a coalgebraic representation of PLP, we need a change of perspective to view a PLP program as a transition system that perform reduction of atoms. Our starting point is the coalgebraic semantics for pure logic programming in Komendantskaya et al. [KMP10], which we briefly recall here. A logic program $\mathbb{L}$ based on At can be read reductively as a system such that given a goal A, $\mathbb{L}$ reduces it to all possible subgoals that one needs to prove in order to prove goal A. We illustrate this perspective with a toy example.

Example 3.1.1. Let logic program $\mathbb{L}$ consist of two clauses, $A \leftarrow B_1$ and $A \leftarrow B_2, B_3$. Then the reductive reading of $\mathbb{L}$ is that, the task of proving atom A can be reduced to that of proving either one of the following sets of atoms: $\{B_1\}$ and $\{B_2, B_3\}$.

Such a reductive reading of logic programs suggests the representation of a propositional logic program $\mathbb{L}$ as a $\mathcal{P}_f\mathcal{P}_f$-coalgebra $\mathsf{l}\colon At \to \mathcal{P}_f\mathcal{P}_f(At)$, where the two $\mathcal{P}_f$ functors have two distinct meanings: the inner $\mathcal{P}_f$ stands for the fact that the bodies of logic program clauses are finite sets of atoms, and the outer $\mathcal{P}_f$ encode the non-deterministic nature of one-step proof search. In the case of Example 3.1.1, these two $\mathcal{P}_f$ correspond to the two sets $\{B_1\}$, $\{B_2, B_3\}$, and the set $\{\{B_1\}, \{B_2, B_3\}\}$, respectively. Intuitively, l reduces an atom A to the set of all finite sets of atoms $\{B_1, \ldots, B_k\}$ satisfying that $A \leftarrow B_1, \ldots, B_k$ is a clause in $\mathbb{L}$. We summarise the central construction for propositional logic programs in Komendantskaya et al. [KMP10].

Definition 3.1.2 ([KMP10])**.** A logic program $\mathbb{L}$ on At determines a $\mathcal{P}_f\mathcal{P}_f$-coalgebra $\mathsf{l}\colon At \to \mathcal{P}_f\mathcal{P}_f(At)$ such that

$$\mathsf{l}(A) = \{\mathsf{body}(\varphi) \mid \mathsf{head}(\varphi) = A, \varphi \in \mathbb{L}\}$$

for arbitrary $A \in At$.

Then the 1-1 correspondence between logic programs and $\mathcal{P}_f\mathcal{P}_f$-coalgebras with finite domains can be concisely expressed as follows.

Definition 3.1.3. The category **PropLP** of propositional logic programs has the following components:

- Objects: each object is a pair $\langle X, \mathbb{L}\rangle$, where X is a finite set, and $\mathbb{L}$ is a propositional logic program using elements in X as atoms.
- Morphisms: a morphism $f\colon \langle X, \mathbb{L}\rangle \to \langle Y, \mathbb{K}\rangle$ is a function $f\colon X \to Y$ that respects the programs $\mathbb{L}$ and $\mathbb{K}$ in the following sense: a clause $B \leftarrow B_1, \ldots B_k$ is in $\mathbb{K}$ if and only if there exists $\mathbb{L}$-clause $A \leftarrow A_1, \ldots, A_\ell$ such that $f(A) = b$ and $f[\{A_1, \ldots, A_\ell\}] = \{B_1, \ldots, B_k\}$.

Proposition 3.1.4. *There is an isomorphism of categories* **PropLP** $\cong Fin(\mathbf{Coalg}(\mathcal{P}_f\mathcal{P}_f))$*, where* $Fin(\mathbf{Coalg}(\mathcal{P}_f\mathcal{P}_f))$ *is the full subcategory of* $\mathbf{Coalg}(\mathcal{P}_f\mathcal{P}_f)$ *with objects restricted to finite sets.*

Proof. For objects, on one hand, every logic program $\mathbb{L}$ on At determines a $Fin(\mathbf{Coalg}(\mathcal{P}_f\mathcal{P}_f))$-object l as in Definition 3.1.2. On the other hand, a coalgebra $\lambda\colon X \to \mathcal{P}_f\mathcal{P}_f(X)$ where X is finite also uniquely defines a logic program $\mathbb{L}$ on X such that for each $A \in X$ and $\{A_1, \ldots, A_k\} \in \lambda(A)$, there is a clause $A \leftarrow A_1, \ldots, A_k$ in $\mathbb{L}$. It is immediate that these two constructions are inverses to each other, thus form a bijection between the objects.

On morphisms, given $\langle X, \mathbb{L}\rangle$ and $\langle Y, \mathbb{K}\rangle$, suppose $f\colon X \to Y$ is a coalgebraic morphism, then the following diagram commutes:

$$\begin{tikzcd}
X \arrow[r, "f"] \arrow[d, "\mathsf{l}"'] & Y \arrow[d, "\mathsf{k}"] \\
\mathcal{P}_f\mathcal{P}_f(X) \arrow[r, "\mathcal{P}_f\mathcal{P}_f(f)"] & \mathcal{P}_f\mathcal{P}_f(Y)
\end{tikzcd}$$

Spelling out the definition, it means that, starting from an arbitrary $A \in X$, suppose $\mathsf{l}(A)$ is the set $\{\{A_1^1, \ldots, A_{k_1}^1\}, \ldots, \{A_1^m, \ldots, A_{k_m}^m\}\}$, then $\mathcal{P}_f\mathcal{P}_f(f) \circ \mathsf{l}(A)$ is

$$\{\{f(A_1^1), \ldots, f(A_{k_1}^1)\}, \ldots, \{f(A_1^m), \ldots, f(A_{k_m}^m)\}\} \tag{3.1}$$

This being equal to $\mathsf{k} \circ f(A)$ means that for each $\{f(A_1^i), \ldots, f(A_{k_i}^i)\}$ from (3.1) is an element in $\mathsf{k}(f(A))$, and vice versa. Thus, $f(A) \leftarrow B_1, \ldots, B_m$ is a clause in $\mathbb{K}$ if and only if its body $\{B_1, \ldots, B_m\}$ is equal to some $\{f(A_1^i), \ldots, f(A_{k_i}^i)\}$. This means that f is a **PropLP**-morphism. The same reasoning works for the other direction showing that a **PropLP**-morphism is also a coalgebra morphism. □

Now we turn to probabilistic logic programs. Compared with the aforementioned intuition for the coalgebraic representation of logic programs, there is one similarity and one difference. For PLP it remains the fact that the bodies of clauses are finite sets of atoms. However, the goal-oriented or proof-searching reading of PLP no longer reduces a goal to a set of subgoals but instead to a set of subgoals augmented with probability labels. From a categorical perspective, this amounts to saying that we represent propositional PLP again as a coalgebra for some composite functor $\mathcal{T}_2 \circ \mathcal{T}_1$: while the inner functor $\mathcal{T}_1$ remains to be the finite powerset functor $\mathcal{P}_f$, the outer functor $\mathcal{T}_2$ should be a combination of finite powerset functor $\mathcal{P}_f$ and probability labels. To be precise, $\mathcal{T}_2$ is the finite multiset functor $\mathcal{M}_{pr}\colon \mathsf{Set} \to \mathsf{Set}$ based on the commutative monoid $([0,1], \vee^{pr}, 0)$, where $\vee^{pr}$ is the 'probabilistic or' defined as $p \vee^{pr} q := 1 - (1-p)(1-q)$, for arbitrary $p, q \in [0,1]$ (see Example 2.2.7 for details). We are now ready to represent propositional PLP programs as $\mathcal{M}_{pr}\mathcal{P}_f$-coalgebras.

Definition 3.1.5. Let $\mathbb{P}$ be a PLP over At. It defines a $\mathcal{M}_{pr}\mathcal{P}_f$-coalgebra $\mathsf{p}\colon At \to \mathcal{M}_{pr}\mathcal{P}_f(At)$, such that for an arbitrary atom $A \in At$,

$$\begin{array}{rrcl} \mathsf{p}(A)\colon & \mathcal{P}_f(At) & \to & [0,1] \\ & \{B_1,\dots,B_k\} & \mapsto & \begin{cases} r & \text{if } r :: A \leftarrow B_1,\dots,B_k \text{ is a clause in } \mathbb{P} \\ 0 & \text{otherwise.} \end{cases} \end{array} \tag{3.2}$$

The map p is a well-defined $\mathcal{M}_{pr}\mathcal{P}_f$-coalgebra: note that each $\mathsf{p}(a)$ has a finite support since program $\mathbb{P}$ consists of only finitely many clauses. Definition 3.1.5 can be equivalently and more concisely expressed using the ket notation $|-\rangle$ (see the notation after Definition 2.2.4) as

$$\mathsf{p}(A) = \sum_{(r::A \leftarrow B_1,\dots,B_n)\ \in\ \mathbb{P}} r|\{B_1,\dots,B_n\}\rangle$$

Example 3.1.6. Consider program $\mathbb{P}^{al}$ from Example 2.1.10. The set of atoms is

$$At_{al} = \{\texttt{Alarm}, \texttt{Earthquake}, \texttt{Burgary}, \texttt{Wake(M)}, \texttt{Paracusia(M)}, \texttt{Hear\_alarm(M)}\} \tag{3.3}$$

Here are some values of the corresponding coalgebra $\mathsf{p}_{al}\colon At_{al} \to \mathcal{M}_{pr}\mathcal{P}_f(At_{al})$, denoted using the ket notation:

$$\begin{aligned} \mathsf{p}_{al}(\texttt{Hear\_alarm(M)}) &= 0.8|\{\texttt{Alarm}, \texttt{Wake(M)}\}\rangle + 0.3|\texttt{Paracusia(M)}\}\rangle \\ \mathsf{p}_{al}(\texttt{Earthquake}) &= 0.01|\{\}\rangle \end{aligned}$$

Similar to the case for logic programs, while (3.2) tells us how to represent every propositional PLP program as a $\mathcal{M}_{pr}\mathcal{P}_f$-coalgebra, the other direction also holds by restricting ourselves to $\mathcal{M}_{pr}\mathcal{P}_f$-coalgebras with finite bases. To state it formally, we introduce the category of propositional PLP programs.

Definition 3.1.7. The *category of propositional PLP programs* **PropPLP** is defined as follows:

- Object: each object is a pair of the form $\langle A, \mathbb{P}\rangle$, where X is a finite set and $\mathbb{P}$ is a PLP program using elements in X as atoms.
- Morphisms: a morphism $f\colon \langle X, \mathbb{P}\rangle \to \langle Y, \mathbb{Q}\rangle$ is a function $f\colon X \to Y$ that respects $\mathbb{P}$ and $\mathbb{Q}$: a clause $p :: B \leftarrow B_1,\dots,B_k$ is in $\mathbb{Q}$ if and only if, suppose $\psi_1,\dots,\psi_m$ are all the $\mathbb{P}$-clauses such that $f(\mathsf{head}(\psi)) = B$ and $f[\mathsf{body}(\psi)] = \{B_1,\dots,B_k\}$, then $p = \mathsf{Lab}(\psi_1) \vee^{pr} \cdots \vee^{pr} \mathsf{Lab}(\psi_m) = 1 - \prod_{i=1}^{m}(1 - \mathsf{Lab}(\psi_i))$.

Then our observation that propositional PLP programs are precisely $\mathcal{M}_{pr}\mathcal{P}_f$-coalgebras with finite bases can be formally states as follows.

Proposition 3.1.8. *There is an isomorphism between categories* **PropPLP** $\cong Fin(\mathbf{Coalg}(\mathcal{M}_{pr}\mathcal{P}_f))$.

Proof. We define two functors $\mathcal{F}\colon \mathbf{PropPLP} \to Fin(\mathbf{Coalg}(\mathcal{M}_{pr}\mathcal{P}_f))$ and $\mathcal{G}\colon Fin(\mathbf{Coalg}(\mathcal{M}_{pr}\mathcal{P}_f)) \to \mathbf{PropPLP}$, and show that they witness the isomorphism.

Let us first look at the objects. Given a **PropPLP**-object $\langle X, \mathbb{P}\rangle$, we define $\mathcal{F}(\langle X, \mathbb{P}\rangle)$ to be the coalgebra $\mathsf{p}\colon X \to \mathcal{M}_{pr}\mathcal{P}_f(X)$ in Definition 3.1.5. Given a $Fin(\mathbf{Coalg}(\mathcal{M}_{pr}\mathcal{P}_f))$ object $\alpha\colon X \to \mathcal{M}_{pr}\mathcal{P}_f(X)$ where X is a finite set, we define the PLP $\mathbb{P}_\alpha$ as follows: for each $A \in X$ and $\{A_1, \dots, A_k\} \in \mathsf{supp}(\alpha(A))$ with $\alpha(A)(\{A_1, \dots, A_k\}) = p$, the clause $p :: A \leftarrow A_1, \dots, A_k$. is in $\mathbb{P}_\alpha$. These two operations are inverses to each other. On one hand, if $p :: A \leftarrow A_1, \dots, A_k$ is a clause in $\mathbb{P}$, then the set $\{A_1, \dots, A_k\}$ is in the support of $\mathsf{p}(A)$, and $\mathsf{p}(A)(\{A_1, \dots, A_k\}) = p$, thus $p :: A \leftarrow A_1, \dots, A_k$ is also in the program induced by p. On the other hand, if $\{A_1, \dots, A_k\} \in \mathsf{supp}(\alpha(A))$ and $\alpha(A)(\{A_1, \dots, A_k\}) = p$, then the clause $p :: A \leftarrow A_1, \dots, A_k$ is in the program $\mathbb{P}_\alpha$ induced by α. Then the coalgebra p_α induced by $\mathbb{P}_\alpha$ satisfies that $\mathsf{p}_\alpha(\{A_1, \dots, A_k\}) = p$, thus it is exactly α.

Next we turn to morphisms. Given a **PropPLP**-morphism $f\colon \langle X, \mathbb{P}\rangle \to \langle Y, \mathbb{Q}\rangle$, we define $\mathcal{F}(f)\colon \mathcal{F}(\langle X, \mathbb{P}\rangle) \to \mathcal{F}(\langle Y, \mathbb{Q}\rangle)$ mapping $A \in X$ to $f(A) \in Y$, and we verify that this $\mathcal{F}(f)$ is a coalgbra morphism between $\mathcal{F}(\langle X, \mathbb{P}\rangle)$ and $\mathcal{F}(\langle Y, \mathbb{Q}\rangle)$, which for simplicity we write as p and q, respectively. In other words, we show that the following diagram commutes:

$$\begin{tikzcd}
X \arrow[r, "f"] \arrow[d, "\mathsf{p}"'] & Y \arrow[d, "\mathsf{q}"] \\
\mathcal{M}_{pr}\mathcal{P}_f(X) \arrow[r, "\mathcal{M}_{pr}\mathcal{P}_f(f)"] & \mathcal{M}_{pr}\mathcal{P}_f(Y)
\end{tikzcd}$$

The commutativity of the above diagram boils down to the following condition: for arbitrary $A \in X$ and set $\{B_1, \dots, B_k\} \subseteq Y$, $\mathsf{q}(f(A))(\{B_1, \dots, B_k\}) = r$ for some $r \in (0, 1]$ if and only if

$$1 - \prod_{\{A_1, \dots, A_m\}} (1 - \mathsf{p}(A)(\{A_1, \dots, A_m\})) = r$$

where the product ranges over all $\{A_1, \dots, A_m\} \in \mathsf{supp}(\mathsf{p}(A))$ such that $f[\{A_1, \dots, A_m\}] = \{B_1, \dots, B_k\}$. But this is exactly the definition that f is a **PropPLP**-morphism, so $\mathcal{F}$ is well-defined on morphisms. Then for the isomorphism, it suffices to show that $\mathcal{F}$ is full and faithful. Note that the condition for coalgebra morphisms boils down exactly to the definition of **PropPLP**-morphisms, thus $\mathcal{F}$ is full. For two different $f, g\colon \langle X, \mathbb{P}\rangle \to \langle Y, \mathbb{Q}\rangle$, suppose $\mathcal{F}(f) = \mathcal{F}(g)$, then by the definition of $\mathcal{F}$, $f(A) = \mathcal{F}(f)(A) = \mathcal{F}(g)(A) = g(A)$ for arbitrary $A \in X$, so $f = g$. This finishes the verification that $\mathcal{F}$ witnesses the bijection on morphisms. Therefore the two categories **PropPLP** and $Fin(\mathbf{Coalg}(\mathcal{M}_{pr}\mathcal{P}_f))$ are isomorphic. □

Remark 3.1.9. One obvious alternative coalgebra type for propositional PLP programs is the functor $\mathcal{P}_f(\mathcal{P}_f(\cdot) \times [0, 1])$, where the inner $\mathcal{P}_f$ encodes the bodies of clauses, and the $[0, 1]$ part keeps track of the probability labels. The main reason why we choose $\mathcal{M}_{pr}\mathcal{P}_f$ instead of this functor to present PLP is that, Proposition 3.1.8 fails for the latter: although every ground PLP program generates a $\mathcal{P}_f(\mathcal{P}_f(\cdot) \times [0, 1])$-coalgebra, the other direction does not hold. This is because intuitively there are more $\mathcal{P}_f(\mathcal{P}_f(\cdot) \times [0,1])$-coalgebras than PLP programs: a clause $\varphi \in \mathcal{P}_f(At)$ may be associated with different probability values in $[0,1]$, which violates the definition PLP. For example, consider $\alpha\colon At \to \mathcal{P}_f(\mathcal{P}_f(At) \times [0,1])$ where $At = \{A, B, C\}$ such that $\alpha(A) = \{(\{B,C\}, 0.3), (\{B,C\}, 0.5)\}$. Then one does not know which clause(s) does $\alpha(A)$ determine.

3.1.2 Derivation Semantics

In this section we are going to define the first coalgebraic semantics of propositional PLP programs. Before the technical developments, we give some intuition on the semantics provided by final coalgebra . The idea is to represent each goal as a *stochastic derivation tree* (Definition 3.1.11), the probabilistic version of so-called *and-or derivation trees* (Definition 3.1.10), which represent parallel SLD-resolutions for pure logic programming [GC94]. These trees turn out to be elements in the final $At \times \mathcal{M}_{pr}\mathcal{P}_f(\cdot)$-coalgebra, and the connection between a goal and its stochastic derivation trees is captured by the final coalgebra semantics.

As before, to get more comfortable with the definition of stochastic derivation trees for PLP, we briefly recall the definition of and-or derivation trees for pure logic programming.

Definition 3.1.10 (And-or derivation trees, [GC94])**.** Given a propositional definite logic program $\mathbb{L}$ based on At, and an atom $A \in At$, the *and-or derivation tree* for a in $\mathbb{L}$ is the possibly infinite tree $\mathcal{T}$ satisfying:

1. Every node is either an *or-node* (labelled by an atom in At) or a *and-node* (labelled with $\bullet$). These two types of nodes appear alternatingly in depth in this order.

2. The root of $\mathcal{T}$ is an or-node labelled with A.

3. Suppose s is an or-node labelled with A', then for each clause $A \leftarrow B_1, \ldots, B_k$ in $\mathbb{L}$, s has exactly one and-child t, and t has k-many or-children labelled with $B_1, \ldots, B_k$, respectively.

Intuitively, an and-or derivation tree for an atom A encodes the parallel proof search of A as a (finite branching) tree. Each or-node represents a single goal, and each and-node represents a set of subgoals. In order to prove an atom A labelling some or-node s, it suffices to prove at least one of the set of subgoals represented by the and-children of t; and to prove the set of subgoals of some and-node t, one needs to prove all the subgoals represented by the or-children of t. This justifies the terminology ‘and-or’: at an or-node at least of one its children should be proven; at an and-node all its children should be proven. In the case of PLP, we want to reflect the reductive nature of PLP using certain stochastic derivation trees, which add to derivation trees probability labels on the edges that corresponds to the probability labels of the clauses. Later in Section 3.1.3 we will encounter a further variation of derivation trees called *possible-world trees* (Definition 3.1.25), which encode the distribution semantics as tree-like structures.

So to keep the terminology uniform, we will use atom-node and clause-node for or-node and and-node, respectively, in the following definition of stochastic derivation trees.

Definition 3.1.11 (Stochastic derivation trees)**.** The *stochastic derivation tree* for atom A in $\mathbb{P}$ is the possibly infinite tree satisfying the following conditions:

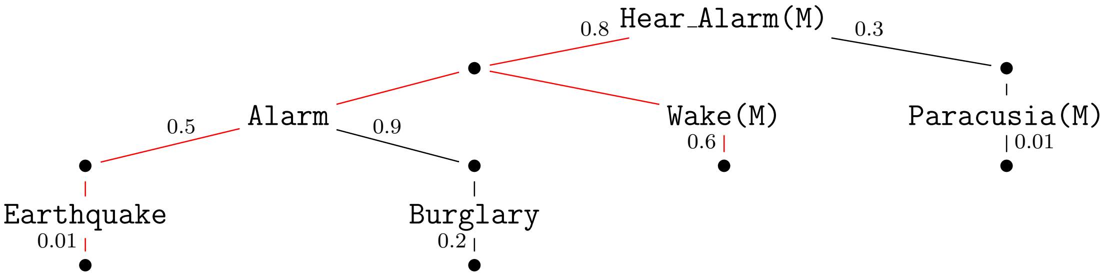
1. Every node is either an *atom-node* (labelled with an atom $B \in At$) or a *clause-node* (labelled with $\bullet$). These two types of nodes appear alternatingly in depth, in this order. In particular, the root is an atom-node labelled with A.

2. Each edge from an atom-node to its (clause-)children is labelled with a probability value.

3. Suppose s is an atom-node with label B. Then for every clause $r :: B \leftarrow B_1, \dots, B_k$ in $\mathbb{P}$, s has exactly one child t such that the edge $s \to t$ is labelled with r, and t has exactly k children labelled with $B_1, \dots, B_k$, respectively.

Note that leaves in stochastic derivation trees can be either atom-nodes or clause-nodes, but they two have different meanings: if an atom-node annotated with B has no child, then there is no clause in $\mathbb{P}$ whose head is B; if a clause-node has no child, then it corresponds to clause with an empty body (*i.e.* a fact). Strictly speaking, an atom A might have multiple and-or (*resp.* stochastic) derivation trees, but these trees differ only in the ordering of the children of each node. Therefore up to permutation (of the children of each single node), there is a unique and-or (*resp.* stochastic) derivation tree for each atom A in program $\mathbb{P}$, denoted as $\mathcal{T}_{\mathbb{P}}(A)$ (*resp.* $\mathcal{T}^{st}_{\mathbb{P}}(A)$). Below we shall use this assumption implicitly in defining distribution trees (*i.e.* Definition 3.1.25). In other words, for derivation trees and distribution trees we use the tree structure where the children of each node form a set instead of a list.

Example 3.1.12. Continuing Example 3.1.6, $[\![\texttt{Hear\_alarm(M)}]\!]_{\mathbb{P}_{al}}$ is the stochastic derivation tree below. The subtree highlighted in red represents one of the successful proofs of `Hear_alarm(M)` in $\mathbb{P}^{al}$: indeed, note that a single child is selected for each atom-node a (corresponding to a clause matching a), all children of any clause-node are selected (corresponding to the atoms in the body of the clause), and the subtree has clause-nodes as leaves (all atoms are proven).



(3.4)

Any such subtree describes a proof, but does not yield a probability value to be associated to a goal — this is the remit of the distribution semantics, see Example 3.1.26.

The final coalgebra semantics $[\![-]\!]_{\mathsf{P}}$ for program $\mathbb{P}$ will map a goal A to the and-or (resp. stochastic) derivation tree representing all possible (resp. stochastic) SLD-resolutions of A in $\mathbb{P}$. Before this we prove the existence of the domain in which these stochastic derivation trees live. It is the cofree coalgebra for $\mathcal{M}_{pr}\mathcal{P}_f$, the dual notion of the more well-known free algebra. Both notions are recalled below.

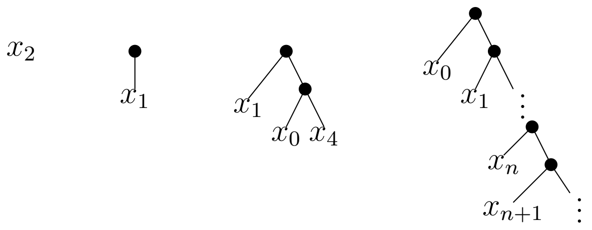
Definition 3.1.13. For a given endofunctor $\mathcal{T}\colon \mathbb{C} \to \mathbb{C}$,

- the *free $\mathcal{T}$-algebra* on object C (if exists) is the initial $C + \mathcal{T}(\cdot)$-algebra, denoted as $\mathfrak{F}(\mathcal{T})(C)$;
- the *cofree $\mathcal{T}$-coalgebra* on object C (if exists) is the final $C \times \mathcal{T}(\cdot)$-coalgebra, denoted as $\mathcal{C}(\mathcal{T})(C)$.

Both notions have very intuitive meaning in the category of sets Set. Given a set X, the free $\mathcal{T}$-algebra on X is the set of all '$\mathcal{T}$-terms' using $x \in X$ as 'free variables'; the free $\mathcal{T}$-coalgebra on X is the set of all '$\mathcal{T}$-trees' whose leaves are $x \in X$.

Example 3.1.14. Free algebras in the category of sets Set are best illustrated using algebraic terms. Let $\Sigma = \{c^0, f^1, g^2\}$ be a signature of function symbols, whose superscripts denote their respective arities. Σ determines an endofunctor $\mathcal{T}$ on Set, namely $\mathcal{T}_\Sigma = 1 + (\cdot) + (\cdot)^2$. Then the free $\mathcal{T}$-algebra on set $X = \{x_0, x_1, \ldots, \}$ is exactly the set of all Σ-terms freely generated using $x_i \in X$ as free variables and Σ. For instance, $\mathfrak{F}(\mathcal{T}_\Sigma)$ contains $x_2, f(f(x_3)), g(fx_1)(f(fc)), \ldots$.

Example 3.1.15. The cofree $\mathcal{P}_f$-coalgebra on $X = \{x_0, x_1, \ldots\}$ are (possibly infinite) trees where each node has a *set* of children instead of a *list* of children, such that every node has finitely many children, and the leaves are $x \in X$. For example, the following trees are all elements in $\mathcal{C}(\mathcal{P}_f)(X)$:



Here the first tree consits of only a single node; the last tree is infinite, where each node has two children and exactly one of these two nodes is a leaf.

The cofree coalgebra — when it exists — can be constructed via a so-called terminal sequence construction, introduced in Worrell [Wor99]. The construction can be seen as a (co-)generalisation of the well-known construction of free algebra as an infinite sum (colimit), as illustrated by Example 3.1.16. To put it simple, by replacing the coproducts with products in the free algebra construction one obtains the cofree coalgebra.

Example 3.1.16. For free algebra, we continue Example 3.1.14. Note that $\mathfrak{F}(\mathcal{T}_\Sigma)(X)$ is equivalently the countable sum $\sum_{i\in\mathbb{N}} \mathcal{T}_\Sigma^i(X)$, where $\mathcal{T}_\Sigma^0(X) = X$. In set-theorectical terms, $\sum_{i\in\mathbb{N}} \mathcal{T}_\Sigma^i(X)$ is exactly the set of all Σ-terms constructed using elements in X as free variables.

Now we present the terminal sequence construction of cofree coalgebra applied to the functor $\mathcal{M}_{pr}\mathcal{P}_f$ for PLP.

Construction 3.1.17. Fix a set Y, the *terminal sequence* for the functor $Y \times \mathcal{M}_{pr}\mathcal{P}_f(\cdot) : \mathsf{Set} \to \mathsf{Set}$ consists of sequences of objects $\{X_\alpha\}_{\alpha\in Ord}$ and arrows $\{\delta_\beta^\alpha \colon X_\alpha \to X_\beta\}_{\beta<\alpha\in Ord}$ constructed by the following transfinite induction:

- For objects,

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

- For morphisms,

$$\delta_\beta^\alpha := \begin{cases} \pi_0 & \alpha = 1, \beta = 0 \\ id_Y \times \mathcal{M}_{pr}\mathcal{P}_f(\delta_\xi^{\xi+1}) & \alpha = \beta + 1 = \xi + 2 \\ \text{the universal map to } X_\beta & \alpha = \beta + 1 \ \& \ \beta \text{ is limit} \\ \text{the limit projections} & \alpha \text{ is limit} \ \& \ \beta < \alpha \end{cases}$$

For the remaining cases where $\alpha = \beta + 1$ for some β and $\xi < \beta$, δ_ξ^α is defined as $\delta_\beta^\alpha \circ \delta_\xi^\beta$.

It is helpful to spell out the induction step for δ in Construction 3.1.17 that involves limit ordinals. For the case where $\alpha = \beta + 1$ and β is a limit ordinal, assume that δ_ξ^α is already defined for all $\xi < \beta$, then $\delta_\alpha^{\alpha+1} \colon X_{\alpha+1} \to X_\alpha$ is the unique morphism derived from the universal property of the limit X_α:

$$\begin{tikzcd}
 & & & X_{\alpha+1} \arrow[dddlll, "\delta_0^{\alpha+1}"'] \arrow[ddll, "\delta_1^{\alpha+1}"'] \arrow[d, dotted, "!"] \arrow[ddr, "\delta_\xi^{\alpha+1}"] \arrow[ddrr, "\cdots"] & & \\
 & & & X_\alpha \arrow[dlll, "\delta_0^\alpha"'] \arrow[dll, "\delta_1^\alpha"'] \arrow[dr, "\delta_\xi^\alpha"] \arrow[drr, "\cdots"] & & \\
Y & Y \times \mathcal{M}_{pr}\mathcal{P}_f(Y) \arrow[l, "\pi_0"] & \cdots \arrow[l, "id\times\mathcal{M}_{pr}\mathcal{P}_f(\pi_0)"'] & & X_\xi \arrow[ll] \arrow[lll, bend left, "\delta_1^\xi"] \arrow[llll, bend left, "\delta_0^\xi"] & \cdots \arrow[l, "\delta_\xi^{\xi+1}"]
\end{tikzcd}$$

For the case where α is a limit ordinal and $\beta \leq \alpha$, note that X_α is defined as the limit $\lim\{\delta_\xi^\chi \mid \xi \leq \chi \leq \alpha\}$, and δ_β^α is defined as the projection morphism $X_\alpha \to X_\beta$.

Proposition 3.1.18 ([Wor99, Corollary 3.3])**.** *If $\mathcal{T}$ is an accessible endofunctor on a locally presentable category that preserves monics, then the terminal $\mathcal{T}$-sequence $\{A_\alpha, f_\beta^\alpha\}$ converges, necessarily to a terminal T-coalgebra.*

Proposition 3.1.19. *The terminal sequence for the functor $Y \times \mathcal{M}_{pr}\mathcal{P}_f(-)$ converges to a limit X_γ such that $X_\gamma \simeq X_{\gamma+1}$.*

Proof. We apply Proposition 3.1.18, and simply verify the conditions for $\mathcal{T}$ there holds for $\mathcal{M}_{pr}\mathcal{P}_f$.

Since Set is a locally presentable category, it remains to verify the two conditions for $\mathcal{M}_{pr}\mathcal{P}_f$. It is a well-known fact that $\mathcal{P}_f$ is ω-accessible, and $\mathcal{M}_{pr}$ has the same property, see e.g. [Sil10, Prop. 6.1.2]. Because accessibility is defined in terms of colimit preservation, it is clearly preserved by composition, and thus $\mathcal{M}_{pr}\mathcal{P}_f$ is also accessible. Then we check that $\mathcal{M}_{pr}\mathcal{P}_f$ preserves monics. For $\mathcal{M}_{pr}$, given any monomorphism $i : C \to D$ in Set, suppose $\mathcal{M}_{pr}(i)(\varphi) = \mathcal{M}_{pr}(i)(\varphi')$ for some $\varphi, \varphi' \in \mathcal{M}_{pr}(C)$. Then for any $d \in D$, $\mathcal{M}_{pr}(i)(\varphi)(d) = \mathcal{M}_{pr}(i)(\varphi')(d)$. If we focus on the image $i[C]$, then there is an inverse function $i^{-1} : i[C] \to C$, and $\mathcal{M}_{pr}(i)(\varphi) = \mathcal{M}_{pr}(i)(\varphi')$ implies that $\varphi(i^{-1}(d)) = \varphi'(i^{-1}(d))$ for any $d \in i[C]$. But this simply means that $\varphi = \varphi'$. As the same is true for $\mathcal{P}_f$ and the property is preserved by composition, we have that $\mathcal{M}_{pr}\mathcal{P}_f$ preserves monics. Therefore we can conclude that the terminal sequence for $Y \times \mathcal{M}_{pr}\mathcal{P}_f(-)$ converges to the cofree $\mathcal{M}_{pr}\mathcal{P}_f$-coalgebra on Y. □

To our purpose, we take the set Y in Construction 3.1.17 and Proposition 3.1.19 to be At, the set of atoms. Note that $X_{\gamma+1}$ is defined as $At \times \mathcal{M}_{pr}\mathcal{P}_f(X_\gamma)$, so the above isomorphism $\delta_\gamma^{\gamma+1} \colon X_{\gamma+1} \to X_\gamma$ induces a coalgebra $(\delta_\gamma^{\gamma+1})^{-1} \colon X_\gamma \to At \times \mathcal{M}_{pr}\mathcal{P}_f(X_\gamma)$, which we denote as ζ. Moreover, this cofree $\mathcal{M}_{pr}\mathcal{P}_f$-coalgebra on At is also the final $At \times \mathcal{M}_{pr}\mathcal{P}_f$-coalgebra. The finality of ζ in the category of $At \times \mathcal{M}_{pr}\mathcal{P}_f$-coalgebras already gives a final coalgebra semantics for an arbitrary $At \times \mathcal{M}_{pr}\mathcal{P}_f$-coalgebra, say $\beta \colon B \to At \times \mathcal{M}_{pr}\mathcal{P}_f(B)$, using the universal morphism $\zeta \to \beta$. But to better understand what this final coalgebra semantics gives us, we need the tree representation of the elements of X_γ, the domain of the final coalgebra ζ. This will finally make the connection back to the resolution tree structure for PLP.

As a slight generalisation of the intuition of cofree $\mathcal{P}_f$-coalgebras from Example 3.1.15, Komendantskaya et al. [KMP10] observed that cofree coalgebras $\mathcal{P}_f\mathcal{P}_f$-coalgebra on At can be seen as and-or trees. In the case of PLP, we replace the first $\mathcal{P}_f$ with $\mathcal{M}_{pr}$ in the coalgebra type, which effectively adds probability values to the edges from and-nodes to or-nodes (which are edges from atom-nodes to clause-nodes in our stochastic derivation trees), as in (3.4). This suggests that stochastic derivation trees as in Definition 3.1.11 are elements of X_γ.

Proposition 3.1.20. *Every stochastic derivation tree is an element in the final $At \times \mathcal{M}_{pr}\mathcal{P}_f(\cdot)$-coalgebra $\zeta \colon X_\gamma \to At \times \mathcal{M}_{pr}\mathcal{P}_f(X_\gamma)$.*

Proof. By the terminal sequence construction of cofree $\mathcal{M}_{pr}\mathcal{P}_f$-coalgebra in Construction 3.1.17, the elements in the final $At \times \mathcal{M}_{pr}\mathcal{P}_f(\cdot)$-coalgebra ζ can be seen as tree structure, where there are two types of nodes, say type (i) and (ii), appearing alternatingly in depth in this order. Every type (i) node is labelled with an atom, with finitely many (possibly none) type (ii) nodes as children; also, each edge from a type (i) node to its child has a label in $(0, 1]$. Every type

(ii) node has finitely many type (i) node as children. Moreover, two siblings has isomorphism subtrees. Then it is immediate to see that every derivation tree (Definition 3.1.11) has such tree structure, where the nodes of types (i) and (ii) are the atom-nodes and clause-nodes, respectively. □

Example 3.1.21. We illustrate the action of the coalgebra map $\zeta\colon X_\gamma \to At \times \mathcal{M}_{pr}\mathcal{P}_f(X_\gamma)$ with the tree $\mathcal{T}$ in (3.4). As an element of X_γ, $\mathcal{T}$ is mapped to the pair $\langle \texttt{Hear\_alarm(M)}, \varphi\rangle$, where $\varphi \in \mathcal{M}_{pr}\mathcal{P}_f(X_\gamma)$ assigns 0.8 to the set consisting of the subtrees of $\mathcal{T}$ with root `Alarm` and with root `Wake(M)`, 0.3 to the singleton consisting to the subtree of $\mathcal{T}$ with root `Paracusia(M)`, and 0 to any other finite set of trees.

Remark 3.1.22. Note that the converse of Proposition 3.1.20 is not true, namely not every tree in the final $At \times \mathcal{M}_{pr}\mathcal{P}_f(\cdot)$-coalgebra is a stochastic derivation tree. Compared with the conditions in Definition 3.1.11, those trees in the final coalgebra only meet conditions (1) and (2), but do not necessarily satisfy (3), which is specific to the structure of the logic program.

With all the definitions at hand, we are now ready to define our first coalgebraic semantics of propositional PLP programs.

Definition 3.1.23. The coalgebraic semantics $[\![-]\!]_{\mathsf{p}}\colon At \to X_\gamma$ of program $\mathbb{P}$ is defined as the unique morphism from the $At \times \mathcal{M}_{pr}\mathcal{P}_f(\cdot)$-coalgebra $\langle id, \mathsf{p}\rangle$ to the final coalgebra $\zeta\colon X_\gamma \to At \times \mathcal{M}_{pr}\mathcal{P}_f(X_\gamma)$. Diagrammatically, the situation is depicted as follows.

$$\begin{tikzcd}
At \arrow[r, dotted, "{[\![-]\!]_{\mathsf{p}}}"] \arrow[d, "{\langle id,\mathsf{p}\rangle}"'] & X_\gamma \arrow[d, "\zeta"] \\
At \times \mathcal{M}_{pr}\mathcal{P}_f(At) \arrow[r, "{id\times\mathcal{M}_{pr}\mathcal{P}_f([\![-]\!]_{\mathsf{p}})}"] & At \times \mathcal{M}_{pr}\mathcal{P}_f(X_\gamma)
\end{tikzcd} \tag{3.5}$$

In particular, $[\![-]\!]_{\mathsf{p}}$ maps each atom $a \in At$ to its stochastic derivation tree $\mathcal{T}^{st}_{\mathbb{P}}(a)$.

Proposition 3.1.24. *For each atom $A \in At$, $[\![A]\!]_{\mathsf{p}} = \mathcal{T}^{st}_{\mathbb{P}}(A)$.*

Proof. By the terminal sequence construction and Proposition 3.1.20, it suffices to show that for each layer the statement is true, namely $[\![A]\!]_{\mathsf{p}}$ and $\mathcal{T}^{st}_{\mathbb{P}}(A)$ coincide on the roots, the roots' children and grandchildren. But this is already implied by the commuting diagram (3.5). □

3.1.3 Distribution semantics

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

In order to connect the distribution semantics with the tree-like derivation semantics of PLP in Subsection 3.1.2, we need to first encode the information of possible worlds and their probabilities in tree-shape. Roughly speaking, a distribution over the sub-programs of the whole program can be formed in a modular way: first we divide the whole program into several pieces, then the original distribution can be seen as the composition of several distributions, precisely one over the sub-programs of each of these pieces. This suggests forming a distribution over the sub-programs along the derivation trees, where at each atom node annotated with $A \in At$ one picks the sub-program of $\mathbb{P}$ whose heads are A. These sub-programs form the aforementioned pieces of the programs. This suggest the following notion of *distribution trees.*

Definition 3.1.25 (Distribution trees)**.** The *distribution tree* for an atom A in program $\mathbb{P}$ is the possibly infinite tree $\mathcal{T}^{dst}_{\mathbb{P}}(A)$ satisfying the following properties:

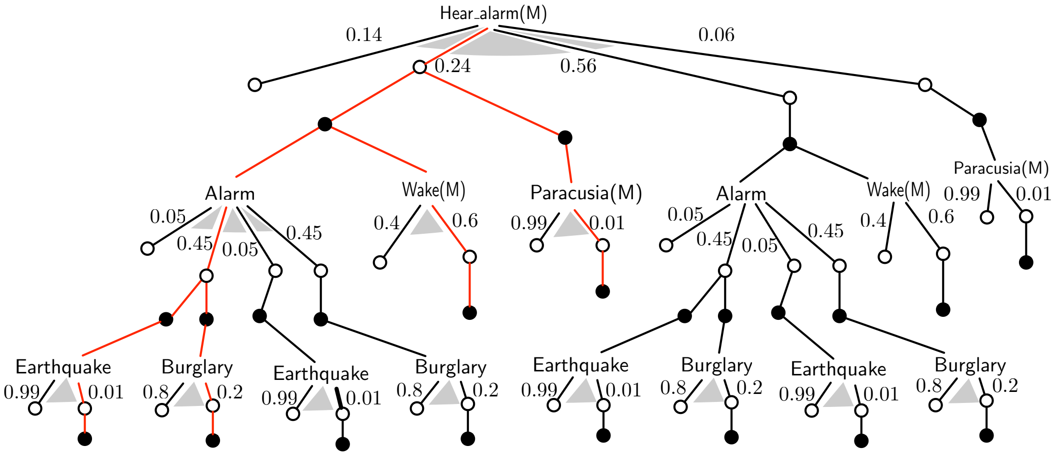
1. Every node is exactly one of the three kinds: atom-node (labelled with an atom $B \in At$), world-node (labelled with $\circ$), clause-node (labelled with $\bullet$). They appear alternatingly in this order in depth. In particular, the root is an atom-node labelled with A.

2. Every edge from an atom-node to its (world-)children is labelled with a probability value, and the labels on all the edges (if exist) from the same atom-node sum up to 1.

3. Suppose s is an atom-node labelled with $A' \in At$, and let $C = \{\varphi_1, \dots, \varphi_m\}$ be the set of all the $\mathbb{P}$-clauses whose heads are A'. Then node s has 2^m (world-)children, each standing for one subset X of C: if a child t stands for X, then the edge $s \to t$ is labelled with probability $\prod_{\varphi \in X} lab(\varphi) \cdot \prod_{\varphi' \in C \setminus X}(1 - lab(\varphi'))$ — recall that $lab(\varphi)$ is the probability labelling φ. Also, t has exactly $|X|$-many (clause-)children, each standing for a clause $\varphi \in X$: if a child u stands for $\varphi = r :: A' \leftarrow B_1, \dots, B_k$, then node u has k (atom-)children, labelled with $B_1, \dots, B_k$ respectively.

Note that the leaves of distribution trees must be either an atom-node or a clause-node, but never a world-node. Comparing distribution trees with stochastic derivation trees (Definition 3.1.11), the former have in addition another class of nodes — world-nodes — that represent sub-programs or possible worlds . Moreover, the possible worlds associated with an atom-node must form a probability distribution, while in stochastic derivation trees probabilities labelling parallel edges do not need to share any relationship. Indeed the probability labels in stochastic derivation trees are seen as the probabilities of independent events. We provide an example of the distribution tree in continuation of our leading example (Examples 2.1.10 and 3.1.12).

Example 3.1.26. In the context of Example 2.1.10, the distribution tree of `Hear_alarm(M)` is depicted below, where we use grey shades to emphasise sets of edges expressing a probability distribution. Also, note the $\circ$s with no children, standing for empty worlds (namely worlds containing no clause). In terms of PLP, they are labelled exactly by atoms matching no clause in the program.



.

In the literature, the distribution semantics usually associates with a goal a single probability value (2.1), rather than a tree structure. However, given the distribution tree it is straightforward to compute such probability. The subtree highlighted in red above describes a refutation of `Hear_alarm(M)`, and the probability of this subtree is the product of all the probabilities appearing in the subtree, namely 0.000001296. The distribution semantics is then the sum of all the probabilities associated to such 'proof' subtrees. The computation is shown in detail in 3.2.4.

So far we have defined distribution trees and seen how it encode the caonnical distribution semantics in tree-like structure. Next we focus on the coalgebraic characterisation of distribution trees and the associated semantics map, similar to the development in Subsection 3.1.2 for stochastic derivation trees and the final coalgebraic semantics $[\![-]\!]$.

Our strategy will be to introduce a novel coalgebra type $\mathcal{D}_{\leq}\mathcal{P}_f\mathcal{P}_f$, such that distribution trees can be seen as elements of the cofree $\mathcal{D}_{\leq}\mathcal{P}_f\mathcal{P}_f$-coalgebra on At, or in other words, elements of the final $At \times \mathcal{D}_{\leq}\mathcal{P}_f\mathcal{P}_f(\cdot)$-coalgebra. Then, we will provide a natural transformation $\mathcal{M}_{pr} \Rightarrow \mathcal{D}_{\leq}\mathcal{P}_f$, which can be used to transform stochastic derivation trees into distribution trees. Finally, composing the universal properties of these cofree coalgebras will yield the desired distribution semantics.

We begin with explaining the intuition behind choosing the coalgebra type $\mathcal{D}_{\leq}\mathcal{P}_f\mathcal{P}_f$. This functor is simply the composite of the sub-distribution functor $\mathcal{D}_{\leq}$ and two finite powerset functors $\mathcal{P}_f$. Based on the experience of cofree $\mathcal{P}_f$-coalgebras (Example 3.1.15) and cofree $\mathcal{M}_{pr}\mathcal{P}_f$-coalgebras (Proposition 3.1.20), cofree $\mathcal{D}_{\leq}\mathcal{P}_f\mathcal{P}_f$-coalgebras can be seen as trees with three different sorts of nodes, one for each functor among $\mathcal{D}_{\leq}$, $\mathcal{P}_f$, $\mathcal{P}_f$. This intuition matches the structure of distribution trees (Definition 3.1.25).

Remark 3.1.27. Note that we cannot work with full probabilities by replacing the functor $\mathcal{D}_{\leq}$ with $\mathcal{D}$ here: a goal may not match any clause, and there is no distribution (though a sub-distribution) over the empty set. In such a case there is no world in which the goal is provable and its probability in the program is 0.

The next step is to recover distribution trees as the elements of the $\mathcal{D}_{\leq}\mathcal{P}_f\mathcal{P}_f$-cofree coalgebra on At. This goes via a terminal sequence, similarly to the case of $\mathcal{M}_{pr}\mathcal{P}_f$ in the previous section. For convenience we will simply state the terminal sequence construction for set At.

Construction 3.1.28. The *terminal sequence* for the functor $At \times \mathcal{D}_{\leq}\mathcal{P}_f\mathcal{P}_f(\cdot) : \mathsf{Set} \to \mathsf{Set}$ consists of sequences of objects $\{Y_\alpha\}_{\alpha \in Ord}$ and arrows $\{\lambda^\alpha_\beta \colon Y_\alpha \to Y_\beta\}_{\beta<\alpha\in Ord}$ constructed by the following transfinite induction:

- For objects,

$$Y_\alpha := \begin{cases} At & \alpha = 0 \\ At \times \mathcal{D}_{\leq}\mathcal{P}_f\mathcal{P}_f(Y_\xi) & \alpha = \xi + 1 \\ \lim\{\lambda^\chi_\xi \mid \xi < \chi < \alpha\} & \alpha \text{ is limit} \end{cases}$$

- For morphisms,

$$\lambda^\alpha_\beta := \begin{cases} \pi_0 & \alpha = 1, \beta = 0 \\ id_{At} \times \mathcal{D}_{\leq}\mathcal{P}_f\mathcal{P}_f(\lambda^{\xi+1}_\xi) & \alpha = \beta + 1 = \xi + 2 \\ \text{the universal map to } Y_\beta & \alpha = \beta + 1 \ \&\ \beta \text{ is limit} \\ \text{the limit projections} & \alpha \text{ is limit } \&\ \beta < \alpha \end{cases}$$

For the remaining cases where $\alpha = \beta + 1$ for some β and $\xi < \beta$, λ^α_ξ is defined as $\lambda^\alpha_\beta \circ \lambda^\beta_\xi$.

The details of the cases for λ^α_β is similar to the explanation right after Construction 3.1.17, and we do not repeat here.

Proposition 3.1.29. *The terminal sequence of $At \times \mathcal{D}_{\leq}\mathcal{P}_f\mathcal{P}_f(\cdot)$ converges at some limit ordinal χ, and $(\lambda^{\chi+1}_\chi)^{-1} : Y_\chi \to At \times \mathcal{D}_{\leq}\mathcal{P}_f\mathcal{P}_f Y_\chi$ is the final $At \times \mathcal{D}_{\leq}\mathcal{P}_f\mathcal{P}_f$ coalgebra.*

Proof. As for Proposition 3.1.19, by [Wor99, Corollary 3.3] it suffices to show that $\mathcal{D}_{\leq}\mathcal{P}_f$ is accessible and preserves monos. Both are simple exercises; in particular, see [BSdV04] for accessibility of $\mathcal{D}_{\leq}$. □

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

Proposition 3.1.30. *Every distribution tree for atoms in $\mathbb{P}$ is an element in Y_χ.*

Proof. The proof is similar to that for stochastic derivation tree (Proposition 3.1.20). In particular, elements in Y_χ can be seen as trees with three types of nodes, which instantiate to atom-nodes, world-nodes, clause-nodes in distribution trees. □

Finally we establish the coalgebraic semantics in terms of distribution trees. For this purpose we use the stochastic derivation trees and its corresponding coalgebraic semantics as an intermediate step, by translating stochastic derivation trees into distribution trees. Graphically, starting from a stochastic derivation tree, for each atom node s one can form a distribution over the subsets of all its children, by regarding the probability labels on the edges to these children nodes as the probability of individual events. The result will share the structure of a distribution tree. To formalise this translation, we introduce a natural transformation $pw : \mathcal{M}_{pr} \Rightarrow \mathcal{D}_{\leq}\mathcal{P}_f$, which basically expands a finite set of independent random events into a joint distribution over the possible outcomes of these events.

Definition 3.1.31. The '**p**ossible **w**orlds' natural transformation $pw \colon \mathcal{M}_{pr} \Rightarrow \mathcal{D}_{\leq}\mathcal{P}_f$ is defined at each set X as

$$pw_X \colon \varphi \mapsto \sum_{Y \subseteq \mathsf{supp}(\varphi)} r_Y |Y\rangle \tag{3.6}$$

where each $r_Y = \prod_{y \in Y} \varphi(y) \cdot \prod_{y' \in \mathsf{supp}(\varphi) \setminus Y} (1 - \varphi(y'))$. In particular, when $\mathsf{supp}(\varphi)$ is empty, $pw_X(\varphi)$ is the empty sub-distribution $\varnothing$.

Proposition 3.1.32. *The class of mappings pw defined in Definition 3.1.31 is a natural transformation.*

Proof. It is straightforward to see that $pw_X(\varphi)$ in (3.6) is an element in $\mathcal{D}_{\leq}\mathcal{P}_f(X)$: $\mathsf{supp}(\varphi)$ is finite, and when $\mathsf{supp}(\varphi)$ is nonempty, $\sum_{Y \subseteq \mathsf{supp}(\varphi)} r_Y = 1$ thus forms a distribution.

To check the naturality, we show that for an arbitrary function $h \colon X \to Y$, the following diagram commutes:

$$\begin{tikzcd}
\mathcal{M}_{pr}(X) \arrow[r, "\mathcal{M}_{pr}(h)"] \arrow[d, "pw_X"'] & \mathcal{M}_{pr}(Y) \arrow[d, "pw_Y"] \\
\mathcal{D}_{\leq}\mathcal{P}_f(X) \arrow[r, "\mathcal{D}_{\leq}\mathcal{P}_f(h)"'] & \mathcal{D}_{\leq}\mathcal{P}_f(Y)
\end{tikzcd} \tag{3.7}$$

Let us start from an arbitrary $\varphi \in \mathcal{M}_{pr}(X)$. The case where φ is an empty probabilistic multiset is trivial, so we assume that φ is nonempty, say $\varphi = p_1|x_1\rangle + \cdots + p_k|x_k\rangle$.

On one hand we follow the down-right route. The distribution $pw_X(\varphi)$ has support $\mathcal{P}(\mathsf{supp}(\varphi))$, and for any $A \in \mathcal{P}(\mathsf{supp}(\varphi))$, $pw_X(\varphi)(A) = \left(\prod_{a \in A} \varphi(a)\right) \cdot \left(\prod_{a \in \mathsf{supp}(\varphi) \setminus A} (1 - \varphi(a))\right)$. Then the support of $\mathcal{D}_{\leq}\mathcal{P}_f(h)(pw_X(\varphi))$ is $h[\mathsf{supp}(pw_X(\varphi))] = h[\mathcal{P}(\mathsf{supp}(\varphi))]$, and for each $B \in h[\mathcal{P}(\mathsf{supp}(\varphi))]$,

$$\mathcal{D}_{\leq}\mathcal{P}_f(h)(pw_X(\varphi))(B) = \sum_{\substack{A \in \mathsf{supp}(pw_X(\varphi)) \\ \&\ h[A] = B}} pw_X(\varphi)(A)$$

$$= \sum_{\substack{A \in \mathcal{P}(\mathsf{supp}(\varphi)) \\ \&\ h[A]=B}} pw_X(\phi)(A)$$

$$= \sum_{\substack{A \in \mathcal{P}(\mathsf{supp}(\varphi)) \\ \&\ h[A]=B}} \left(\prod_{a \in A} \varphi(a) \cdot \prod_{a \in \mathsf{supp}(\varphi) \setminus A} (1 - \varphi(a)) \right) \tag{3.8}$$

On the other hand we follow the right-down route. $\mathcal{M}_{pr}(h)(\varphi)$ has support $h[\mathsf{supp}(\varphi)]$, and for arbitrary $y \in h[\mathsf{supp}]$, $\mathcal{M}_{pr}(h)(\varphi)(y) = \bigvee^{pr}_{x \in h^{-1}(y)} \varphi(x) = 1 - \prod_{x \in h^{-1}(y)} (1 - \varphi(x))$. Then $pw_Y(\mathcal{M}_{pr}(h)(\varphi))$ has support $\mathcal{P}(\mathsf{supp}(\mathcal{M}_{pr}(h)(\varphi))) = \mathcal{P}(h[\mathsf{supp}(\varphi)])$. And for each B' in this support set,

$$pw_Y(\mathcal{M}_{pr}(h)(\varphi))(B') = \prod_{y \in B'} \mathcal{M}_{pr}(h)(\varphi)(y) \cdot \prod_{y \in h[\mathsf{supp}(\varphi)] \setminus B'} (1 - \mathcal{M}_{pr}(h)(\varphi)(y)) \tag{3.9}$$

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

Next we further expand (3.9) by spelling out the definition of $\mathcal{M}_{pr}(h)$. We leave the detailed explanation to steps (3.10), (3.11) below till after the calculation.

$$\begin{aligned}
& pw_Y(\mathcal{M}_{pr}(h)(\varphi))(B') \\
&= \prod_{y \in B'} \left(1 - \prod_{x \in h^{-1}(y)} (1 - \varphi(x)) \right) \cdot \prod_{y \in h[\mathsf{supp}(\varphi)] \setminus B'} \left(1 - \left(1 - \prod_{x \in h^{-1}(y)} (1 - \varphi(x)) \right) \right) \\
&= \prod_{y \in B'} \left(1 - \prod_{x \in h^{-1}(y)} (1 - \varphi(x)) \right) \cdot \prod_{y \in h[\mathsf{supp}(\varphi)] \setminus B'} \prod_{x \in h^{-1}(y)} (1 - \varphi(x)) \\
&\overset{(3.12)}{=} \prod_{y \in B'} \left(\sum_{Z \subseteq_i h^{-1}(y)} \left(\prod_{x \in Z} \varphi(x) \cdot \prod_{x \in h^{-1}(y) \setminus Z} (1 - \varphi(x)) \right) \right) \cdot \prod_{\substack{y \in h[\mathsf{supp}(\varphi)] \setminus B' \\ x \in h^{-1}(y)}} (1 - \varphi(x)) && (3.10) \\
&= \sum_{\substack{A' \subseteq \mathsf{supp}(\varphi) \\ \&\ h[A']=B'}} \left(\prod_{x \in A'} \varphi(x) \cdot \prod_{x \in h^{-1}[B] \setminus A'} (1 - \varphi(x)) \right) \cdot \prod_{\substack{y \in h[\mathsf{supp}(\varphi)] \setminus B' \\ x \in h^{-1}(y)}} (1 - \varphi(x)) && (3.11) \\
&= \sum_{\substack{A' \subseteq \mathsf{supp}(\varphi) \\ \&\ h[A']=B'}} \left(\prod_{x \in A'} \varphi(x) \cdot \prod_{x \in h^{-1}[B'] \setminus A'} (1 - \varphi(x)) \right) \cdot \prod_{\substack{x \in \mathsf{supp}(\varphi) \\ \&\ x \notin h^{-1}[B']}} (1 - \varphi(x)) \\
&= \sum_{\substack{A' \subseteq \mathsf{supp}(\varphi) \\ \&\ h[A']=B'}} \left(\prod_{x \in A'} \varphi(x) \cdot \prod_{x \in h^{-1}[B'] \setminus A'} (1 - \varphi(x)) \cdot \prod_{\substack{x \in \mathsf{supp}(\varphi) \\ x \notin h^{-1}[B']}} (1 - \varphi(x)) \right)
\end{aligned}$$

$$= \sum_{\substack{A' \subseteq \mathsf{supp}(\varphi) \\ \& \ h[A']=B'}} \left(\prod_{x \in A'} \varphi(x) \cdot \prod_{x \in \mathsf{supp}(\varphi) \setminus A'} (1 - \varphi(x)) \right)$$

$$\overset{(3.8)}{=} \mathcal{D}_{\leq}\mathcal{P}_f(h)(\mathcal{M}_{pr}(\varphi))(B')$$

The two steps (3.10) and (3.11) in the above reasoning deserves more explanation. In (3.10), $\subseteq_i$ is the subset relation $\subseteq$ restricted to non-empty subsets, so the sum is over all non-empty subsets Z of $h^{-1}(y)$. Moreover, (3.10) instantiates the following more general equation that holds for arbitrary set X, a probabilistic multiset $\psi \in \mathcal{M}_{pr}(X)$ and set $A \subseteq \mathsf{supp}(\psi)$:

$$1 - \prod_{a \in A}(1 - \psi(a)) = \sum_{Z \subseteq_i A} \left(\prod_{a \in Z} \psi(a) \cdot \prod_{a \in A \setminus Z} (1 - \psi(a)) \right) \tag{3.12}$$

It generalises the equation $1 - (1-p)(1-q) = pq + p(1-q) + q(1-p)$. We briefly explain its proof in words. Intuitively, one can think of A as a (finite) set of independent events, and $\psi(a)$ as the probability of event a. Then the left-hand-side of (3.12) calculates 1 minus the probability that none of $a \in A$ happens; the right-hand-side sums up the probabilities of all possible worlds in which some of $a \in A$ happens. Both calculate the same value, namely the probability that at least one event in A happens.

Equation (3.11) is obtained by expanding $\prod_{y \in B'} \sum_{Z \subseteq_i h^{-1}(y)} G(y, Z)$ into a summation, where $G(y, Z) := \prod_{x \in Z} \varphi(x) \cdot \prod_{x \in h^{-1}(y) \setminus Z}(1 - \varphi(x))$. In the resulting summation, each summand chooses exactly one summand from every sum $\sum_{Z \subseteq_i h^{-1}(y)} G(y, Z)$ for a given $y \in B'$. Moreover, every such choice corresponds to exactly one A' satisfying $h[A'] = B'$: basically we select for each $y \in B'$ a non-empty subset of pre-images of y. So we have

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

and this is exactly the summation in (3.11). □

Now the aformentioned intuition for translating stochastic derivation trees into distribution trees can be formalised using pw. More precisely, applying pw at each atom node in the stochastic trees returns the desired derivation tree, and such 'layer-by-layer' transformation can be formalised using the universal property of final coalgebras.

Definition 3.1.33. The coalgebraic distribution semantics $\langle\!\langle - \rangle\!\rangle_{\mathsf{p}}$ is defined as the following unique morphism $At \to \mathcal{C}(\mathcal{D}_{\leq}\mathcal{P}_f\mathcal{P}_f)$.

$$\begin{tikzcd}
At \arrow[r, dashed, "\llbracket - \rrbracket_{\mathsf{p}}"] \arrow[rr, dashed, bend left, "\langle\!\langle - \rangle\!\rangle_{\mathsf{p}}"] \arrow[d, "<id_{At}{,}p>"'] & X_\gamma \arrow[r, dashed, "ex"] \arrow[d, "\cong"] & Y_\chi \arrow[dd, "\cong"] \\
At \times \mathcal{M}_{pr}\mathcal{P}_f At \arrow[r, "id_{At}\times\mathcal{M}_{pr}\mathcal{P}_f(\llbracket - \rrbracket_{\mathsf{p}})"] \arrow[d, "id_{At}\times pw_{\mathcal{P}_f(At)}"'] & At \times \mathcal{M}_{pr}\mathcal{P}_f X_\gamma \arrow[d, "id_{At}\times pw_{\mathcal{P}_f(X_\gamma)}"] & \\
At \times \mathcal{D}_{\leq}\mathcal{P}_f\mathcal{P}_f At \arrow[r, "id_{At}\times\mathcal{D}_{\leq}\mathcal{P}_f\mathcal{P}_f(\llbracket - \rrbracket_{\mathsf{p}})"] & At \times \mathcal{D}_{\leq}\mathcal{P}_f\mathcal{P}_f X_\gamma \arrow[r, "id_{At}\times\mathcal{D}_{\leq}\mathcal{P}_f\mathcal{P}_f(ex)"] & At \times \mathcal{D}_{\leq}\mathcal{P}_f\mathcal{P}_f Y_\chi
\end{tikzcd} \tag{3.13}$$

Let us provide some explanation for (3.13). The left-top square is the one defining the coalgebraic semantics $\langle\!\langle - \rangle\!\rangle_{\mathsf{p}}$ in terms of stochastic derivation trees. The left-bottom square applies pw exactly one step, and form a distribution for a possible world of atoms or stochastic derivation trees. The right square and the whole big square both apply the universal property of the final $At \times \mathcal{D}_{\leq}\mathcal{P}_f\mathcal{P}_f(\cdot)$-coalgebra Y_χ. In particular, ex extends probabilistic logic programs and stochastic derivation trees to the same coalgebra type as distribution trees. Then the distribution semantics $\langle\!\langle - \rangle\!\rangle_{\mathsf{p}}$ is defined as the unique morphism to Y_χ. By uniqueness, $\langle\!\langle - \rangle\!\rangle_{\mathsf{p}}$ can also be computed as the composite $ex \circ \llbracket - \rrbracket_{\mathsf{p}}$, that is, first one derives the semantics $\llbracket - \rrbracket_{\mathsf{p}}$, then applies the translation pw to each level of the resulting stochastic derivation tree, in order to turn it into a distribution tree.

Finally, this coalgebraic semantics $\langle\!\langle - \rangle\!\rangle_{\mathsf{p}}$ exactly assigns to each atom its distribution tree.

Proposition 3.1.34. *For each atom $A \in At$, $\langle\!\langle A \rangle\!\rangle_{\mathsf{p}} = \mathcal{T}^{dst}_{\mathbb{P}}(A)$.*

Proof. The proof is similar to that for stochastic derivation trees (Proposition 3.1.24). In particular, the commuting diagram (3.13) implies that at each layer $\langle\!\langle A \rangle\!\rangle_{\mathsf{p}}$ behaves exactly as the distribution tree of A. □

3.2 Probabilistic logic programming: first-order case

In this subsection we generalise our coalgebraic treatment for the propositional case to arbitrary probabilistic logic programs and goals, possibly including variables. First, in Subsection 3.2.1, we give a coalgebraic representation for general PLP, and equip it with a final coalgebra semantics in terms of stochastic derivation trees (Subsection 3.2.2). Next, in Subsection 3.2.3, we study the coalgebraic representation of the distribution semantics. Before the technical development, we begin by introducing our leading example — an extension of Example 2.1.10.

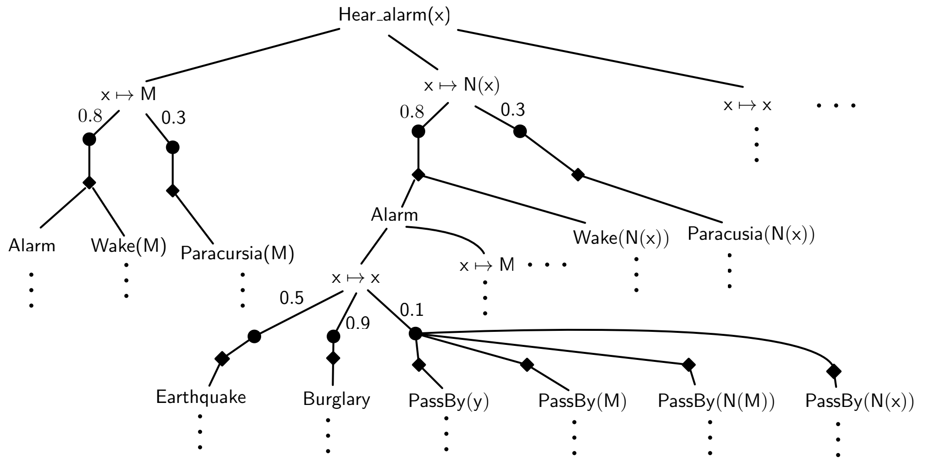
Example 3.2.1. We tweak the ground program of Example 2.1.10 to allow variables and function symbols. Let us explain the scenario alongside looking at the details of the program.

Now it is not just Mary that may hear the alarm, but also her neighbours (represented by $\texttt{Neigh(Mary)}, \texttt{Neigh(Neigh(Mary))}, \ldots$). Moreover, besides the aforementioned two reasons in Example 2.1.10 for the alarm to ring, there is also a small probability that the alarm rings because someone passes too close to Mary's house. Also, we can only estimate the possibility of paracusia and being awake for Mary, not the neighbours (but intuitively these factors shall not affect whether the alarm in Mary's place rings or not). The revised program $\mathbb{P}_{al}$ is based on an extension of the language in Example 2.1.10: we add a new 1-ary function symbol $\texttt{Neigh}$[1] to the signature and a new 1-ary predicate $\texttt{PassBy}(-)$ to the alphabet. Note the appearance of a variable x, which can be freely substituted by an arbitrary term.

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

We attempt to maintain our approach as a direct generalisation of the coalgebraic semantics for pure logic programs in Bonchi and Zanasi [BZ15], which we briefly recall here. There the derivation semantics for general (definite) logic programs represents resolution by *unification.* This means that, at each step of the computation, given a goal A, one seeks substitutions θ, τ such that $A\theta = H\tau$ for the head H of some clause in the program.

Example 3.2.2. In the context of Example 3.2.1, the tree for $[\![\texttt{Hear\_alarm(x)}]\!]_{\mathbb{P}_{al}}$ is (partially) depicted below. For the reason of space, in (3.14) we abbreviate Neigh as $\texttt{N}$ and $\texttt{Mary}$ as $\texttt{M}$. Compared to the ground case (Example 3.1.12), now substitutions applied on the goal side appear explicitly as labels (*e.g.* $x \mapsto \texttt{N(x)}$). Moreover, note that for each atom $A\theta$ and clause φ, there might be multiple substitutions τ such that $A\theta = \mathsf{head}(\varphi)\tau$. This is presented in the following tree (3.14) by the ⧫-labelled nodes, which are children of the clause-nodes (labelled with •).



(3.14)

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

$$\mathsf{p}(A\theta)\colon \quad \{B_1\tau, \dots, B_k\tau\}_{\tau\in\Gamma} \quad \mapsto \begin{cases} r & (r :: H \leftarrow B_1, \dots, B_k) \in \mathbb{P} \text{ and} \\ & \quad \Gamma \text{ contains all } \tau \text{ s.t. } A\theta = H\tau \\ 0 & \text{otherwise.} \end{cases} \tag{3.15}$$

Intuitively, one application of such map is represented in a tree structure as Example 3.2.2 by the first two layers of the subtree rooted at θ. Note that each element in the support of $\mathsf{p}(A)$ is a *countable* set $\{B_1\tau, \dots, B_k\tau\}_{\tau\in\Gamma}$ of instances of the same body $B_1, \dots, B_k$ is that the same clause may match a goal with countably many different substitutions τ_i. For example in the bottom part of (3.14) there are countably infinite substitutions τ_i matching the head of $\mathtt{Alarm} \leftarrow \mathtt{PassBy(x)}$ to the goal $\mathtt{Alarm}$, substituting x with countably infinitely many constants $\mathtt{Mary}, \mathtt{Neigh(Mary)}, \mathtt{Neigh(x)}, \cdots$. In the next part we will see this reflected in the coalgebraic representation of PLP (see (3.17) below) by the use of the countable powerset functor $\mathcal{P}_c$.

In order to model arbitrary unification, the second step is considering all substitutions θ on the goal A such that a term-matcher for $A\theta$ exists. There is an elegant categorical construction [BZ15] packing together these two steps into a single coalgebra map. We will start with this in the next session. Below we fix an alphabet $\mathcal{A} = \langle \Sigma, \mathit{Var}, \mathit{Pred}\rangle$ and a general PLP program based on $\mathcal{A}$ (see Section 2.1).

3.2.1 Coalgebraic representation of general PLP

Towards a categorification of general PLP, the first concern is to account for the presence of variables in atoms. For this we adopt a standard functorial approach using Lawvere theories. The idea originates from Lawvere's doctorial dessertaion [Law63]: an algebraic theory is defined as certain category with finite products that encode operators, and models of that theory are identified with product-preserving functors from that category to the semantic category of choice. Here we focus on algebraic theories of Σ-atoms and substitution between Σ-terms, thus the following definition of the category of theories.

Definition 3.2.3 (Lawvere theory)**.** The opposite *Lawvere Theory* of Σ is $\mathcal{L}_\Sigma^{op}$, where:

- objects are natural numbers standing for contexts;
- a morphism $n \to m$ is an n-tuple of Σ-terms in context m, namely $\langle t_1, \dots, t_n\rangle$ where each t_i has free variables appearing in $x_1, \dots, x_m$.

Intuitively, object n in $\mathcal{L}_\Sigma^{op}$ can be thought of as the set of all Σ-atoms in context n, and each morphism $\langle t_1, \dots, t_n\rangle\colon n \to m$ is a substitution — thus we denote as θ when necessary — such that every x_i is replaced with t_i, so when applied to an atom $A(x_1, \dots, x_n)$ in context n

the result is an atom $A[t_1/x_1,\dots,t_n/x_n]$ in context m. This is formalised by letting the space of atoms on an alphabet $\mathcal{A}$ be a presheaf $\mathsf{At}\colon \mathcal{L}_\Sigma^{op} \to \mathsf{Set}$ rather than a set.

Definition 3.2.4. The functor $\mathsf{At}\colon \mathcal{L}_\Sigma^{op} \to \mathsf{Set}$ is defined as follows:

- On objects, $\mathsf{At}(n)$ is the set of all atoms in context n.
- Ob morphisms, given an n-tuple $\theta = \langle t_1,\dots,t_n\rangle \in \mathcal{L}_\Sigma^{op}[n,m]$ of Σ-terms in context m, $\mathsf{At}(\theta)\colon \mathsf{At}(n) \to \mathsf{At}(m)$ is defined by substitution, namely $\mathsf{At}(\theta)(A) = A\theta$, for any $A \in \mathsf{At}(n)$.

As observed in Komendantskaya *et al.* [KP11] for pure logic programs, if we naively try to model our specification p for program $\mathbb{P}$ (3.15) as a coalgebra on At, we run into problems: indeed p is not a natural transformation, namely the following naturality diagram does not commute.

$$\begin{tikzcd}
\mathsf{At}(n) \arrow[r, "\mathsf{p}_n"] \arrow[d, "\mathsf{At}(\theta)"'] & \mathcal{P}_c\mathcal{P}_f(\mathsf{At}(n)) \arrow[d, "\mathcal{P}_c\mathcal{P}_f(\mathsf{At}(\theta))"] \\
\mathsf{At}(m) \arrow[r, "\mathsf{p}_m"] & \mathcal{P}_c\mathcal{P}_f(\mathsf{At}(m))
\end{tikzcd}$$

Intuitively, this is because the existence of a term-matching for a substitution instance $A\theta$ of A does not necessarily imply the existence of a term-matching for A itself. For pure logic programs, this problem can be solved in at least two ways. First, Komendantskaya and Power [KP11] relaxes naturality by changing the base category of presheaves from Set to Poset. We take here the second route from Bonchi and Zanasi [BZ15], namely give a 'saturated' coalgebraic treatment of PLP that generalises the modelling of pure logic programs. This approach has the advantage of letting us work with Set-based presheaves, and be still able to recover term-matching via a 'desaturation' operation — see Bonchi and Zanasi [BZ15] and Proposition 3.2.26. A detailed discussion of the relation between these two approaches can be found in Power [Pow16].

The key categorical tool we need for the saturated approach is the following *saturation adjunction.*

Definition 3.2.5 (Saturation adjunction)**.** The *saturation adjunction* $\mathcal{U} \dashv \mathcal{K}$ is the following adjunction on presheaf categories:

$$\begin{tikzcd}
\mathsf{Set}^{\mathcal{L}^{op}} \arrow[r, bend left, "\mathcal{U}"] \arrow[r, phantom, "\perp"] & \mathsf{Set}^{|\mathcal{L}^{op}|} \arrow[l, bend left, "\mathcal{K}"]
\end{tikzcd} \tag{3.16}$$

Here $|\mathcal{L}^{op}|$ is the discretisation of $\mathcal{L}^{op}$, *i.e.* all the arrows but the identities are dropped. The left adjoint $\mathcal{U}$ is the forgetful functor, given by precomposition with the obvious inclusion $\iota\colon |\mathcal{L}_\Sigma^{op}| \to \mathcal{L}_\Sigma^{op}$.

The (necessarily unique) right adjunction $\mathcal{K}$ indeed sends every presheaf $\mathcal{F}\colon |\mathcal{L}_\Sigma^{op}| \to \mathsf{Set}$ to its *right Kan extension* $\mathsf{Ran}_\iota\colon \mathsf{Set}^{|\mathcal{L}^{op}|} \to \mathsf{Set}^{\mathcal{L}_\Sigma^{op}}$ along ι, as in the following diagram:

$$\begin{tikzcd}
|\mathcal{L}^{op}| \arrow[r, hook, "\iota"] \arrow[d, "\mathcal{F}"'] & \mathcal{L}^{op} \arrow[dl, "\mathcal{K}(\mathcal{F})"] \\
\mathsf{Set} &
\end{tikzcd}$$

Intuitively $\mathcal{K}(\mathcal{F})$ is the 'best possible solution' to the above commuting diagram. Moreover, $\mathcal{K}(\mathcal{F})$ as a right Kan extension can be computed following Mac Lane [Mac71] as follows:

- On objects, given a functor $\mathcal{F} \in \mathsf{ob}(\mathsf{Set}^{\mathcal{L}_\Sigma^{op}})$, the presheaf $\mathcal{K}(\mathcal{F})\colon \mathcal{L}_\Sigma^{op} \to \mathsf{Set}$ is defined by letting $\mathcal{K}(\mathcal{F})(n)$ be the product $\prod_{\theta \in \mathcal{L}^{op}[n,m]} \mathcal{F}(m)$, where m ranges over $\mathsf{ob}(\mathcal{L}_\Sigma^{op})$. Intuitively, every element in $\mathcal{K}(\mathcal{F})(n)$ is a tuple with index set $\bigcup_{m \in \mathsf{ob}(\mathcal{L}_\Sigma^{op})} \mathcal{L}_\Sigma^{op}[n,m]$, and its component at index $\theta\colon n \to m$ must be an element in $\mathcal{F}(m)$. We follow the convention of [BZ15] and write $\dot{x}, \dot{y}, \dots$ for such tuples, and $\dot{x}(\theta)$ for the component of $\dot{x}$ at index θ.

  With this convention, given an arrow $\sigma \in \mathcal{L}_\Sigma^{op}[n,n']$, $\mathcal{K}(\mathcal{F})(\sigma)$ is defined by pointwise substitution as the mapping of the tuple $\dot{x}$ to the tuple $\langle \dot{x}(\theta \circ \sigma)\rangle_{\theta\colon n' \to m}$.

- On arrows, given a morphism $\alpha\colon \mathcal{F} \to \mathcal{G}$ in $\mathsf{Set}^{|\mathcal{L}_\Sigma^{op}|}$, $\mathcal{K}(\alpha)$ is a natural transformation $\mathcal{K}(\mathcal{F}) \to \mathcal{K}(\mathcal{G})$ defined pointwisely as $\mathcal{K}(\alpha)(n)\colon \dot{x} \mapsto \langle \alpha_m(\dot{x}(\theta))\rangle_{\theta\colon n \to m}$.

It is also useful to record the unit $\eta^{\mathcal{KU}}\colon 1 \Rightarrow \mathcal{KU}$ of the adjunction $\mathcal{U} \dashv \mathcal{K}$. Given a presheaf $\mathcal{F}\colon \mathcal{L}_\Sigma^{op} \to \mathsf{Set}$, $\eta_{\mathcal{F}}^{\mathcal{KU}}\colon \mathcal{F} \Rightarrow \mathcal{KUF}$ is a natural transformation, such that at each $n \in \mathbb{N}$, $\eta_{\mathcal{F}}(n)\colon \mathcal{F}(n) \to \mathcal{KUF}(n)$ is defined by $\eta_{\mathcal{F}}(n)\colon x \mapsto \langle \mathcal{F}(\theta)(x)\rangle_{\theta\colon n \to m}$.

We are now ready to construct the coalgebra structure on the presheaf At to model PLP. First, we are now able to represent p in (3.15) as a coalgebra morphism. The aforementioned naturality issue is solved by defining p as a morphism in $\mathsf{Set}^{|\mathcal{L}_\Sigma^{op}|}$ rather than in $\mathsf{Set}^{\mathcal{L}_\Sigma^{op}}$. Indeed naturality now becomes trivial. The coalgebra p (over $\mathcal{U}\mathsf{At}$) will now have the following type

$$\mathsf{p}\colon \mathcal{U}\mathsf{At} \to \widehat{\mathcal{M}_{pr}}\widehat{\mathcal{P}_c}\widehat{\mathcal{P}_f}\mathcal{U}\mathsf{At} \tag{3.17}$$

where $\widehat{(\cdot)}$ is the obvious extension of Set-endofunctors to $\mathsf{Set}^{|\mathcal{L}_\Sigma^{op}|}$-endofunctors, defined by functor precomposition. Take $\mathcal{P}_f$ as an example, the lifted functor $\widehat{\mathcal{P}_f}\colon \mathsf{Set}^{|\mathcal{L}_\Sigma^{op}|} \to \mathsf{Set}^{|\mathcal{L}_\Sigma^{op}|}$ maps $F\colon |\mathcal{L}_\Sigma^{op}| \to \mathsf{Set}$ to $|\mathcal{L}_\Sigma^{op}| \xrightarrow{F} \mathsf{Set} \xrightarrow{\mathcal{P}_f} \mathsf{Set}$.

With respect to the ground case, note the insertion of $\widehat{\mathcal{P}_c}$, the lifting of the *countable* powerset functor, in order to account for the countably many instances of a clause that may match the given goal (*cf.* the discussion below (3.15)).

Example 3.2.6. Our program $\mathbb{P}_{al}$ from Example 3.2.1 is based on $At_{al}\colon \mathcal{L}_\Sigma^{op} \to \mathsf{Set}$. Some of its values are listed below:

$$At_{al}(0) = \{\mathtt{Mary}, \mathtt{Neigh}(\mathtt{Mary}), \mathtt{Neigh}(\mathtt{Neigh}(\mathtt{Mary})), \dots\}$$
$$At_{al}(1) = \{\mathtt{x}, \mathtt{Mary}, \mathtt{Neigh}(\mathtt{x}), \mathtt{Neigh}(\mathtt{Mary}), \dots\}$$

Part of the coalgebra p_{al} modelling the program $\mathbb{P}^{al}$ is as follows (*cf.* the tree (3.14)). For space reason, we abbreviate `Mary` as `M` in the second equation.

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

We are now ready to define the coalgebra performing unification, rather than just term-matching. Essentially $\mathcal{K}$ enables one to

Definition 3.2.7. Every PLP program $\mathbb{P}$ is represented as a $\mathcal{K}\widehat{\mathcal{M}_{pr}}\widehat{\mathcal{P}_c}\widehat{\mathcal{P}_f}\mathcal{U}$-coalgebra $\mathsf{p}^\sharp$ on At, defined as

$$
\mathsf{p}^\sharp := \mathsf{At} \xrightarrow{\eta_{\mathsf{At}}} \mathcal{K}\mathcal{U}\mathsf{At} \xrightarrow{\mathcal{K}\mathsf{p}} \mathcal{K}\widehat{\mathcal{M}_{pr}}\widehat{\mathcal{P}_c}\widehat{\mathcal{P}_f}\mathcal{U}\mathsf{At} \tag{3.18}
$$

where $\eta^{\mathcal{K}\mathcal{U}}$ is the unit of the saturation adjunction $\mathcal{U} \dashv \mathcal{K}$.

Spelling the definition out, at each $n \in \mathbb{N}$, $\mathsf{p}^\sharp$ is the mapping

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

Intuitively, $\mathsf{p}^\sharp_n$ retrieves all the unifiers $\langle \theta, \tau \rangle$ of A and heads H in $\mathbb{P}$: for each $\mathcal{L}_\Sigma^{op}$-morphism $\theta \colon n \to m$, first we have $A\theta \in At(m)$ as a component of the saturation of A by $\eta^{\mathcal{K}\mathcal{U}}_{\mathsf{At}}$; then we term-match the heads of $\mathbb{P}$-clause with $A\theta$ by p_m.

Remark 3.2.8. Note that the parameter $n \in \mathsf{ob}(\mathcal{L}_\Sigma^{op})$ in the natural transformation $\mathsf{p}^\sharp$ fixes the pool $\{x_1, \dots, x_n\}$ of variables appearing in the atoms (and relative substitutions) that are considered in the computation. Analogously to the case of pure logic programs [KP11, BZ15], it is intended that such n can always be chosen 'big enough' so that all the relevant substitution instances of the current goal and clauses in the program are covered — note the variables occurring therein always form a *finite* set, included in $\{x_1, \dots, x_m\}$ for some $m \in \mathbb{N}$.

3.2.2 Derivation Semantics

Once we have identified our coalgebra type, the construction leading to the derivation semantics $[\![-]\!]_{\mathsf{p}^\sharp}$ for general PLP is completely analogous to the ground case. One can define the cofree coalgebra for $\mathcal{K}\widehat{\mathcal{M}_{pr}}\widehat{\mathcal{P}_c}\widehat{\mathcal{P}_f}\mathcal{U}$ by terminal sequence, similarly to Construction 3.1.17. For simplicity, henceforth we denote the functor $\mathcal{K}\widehat{\mathcal{M}_{pr}}\widehat{\mathcal{P}_c}\widehat{\mathcal{P}_f}\mathcal{U}(-)$ by $\mathcal{S}$.

Construction 3.2.9. The terminal sequence for $\mathsf{At} \times \mathcal{S}(\cdot) : \mathsf{Set}^{\mathcal{L}^{op}} \to \mathsf{Set}^{\mathcal{L}^{op}}$ consists of a sequence of objects X_α and morphisms $\delta^\beta_\alpha \colon X_\beta \to X_\alpha$, for $\alpha < \beta \in Ord$, defined analogously to Construction 3.1.17, with $\mathsf{p}^\sharp$ and $\mathcal{S}$ replacing p and $\mathcal{F}\mathcal{P}_f$, respectively.

This terminal sequence converges by the following proposition.

Proposition 3.2.10. *$\mathcal{S}$ is accessible, and preserves monomorphisms.*

Proof. Since both properties are preserved by composition, it suffices to show that they hold for all the component functors. For $\widehat{\mathcal{M}_{pr}}$, $\widehat{\mathcal{P}_c}$ and $\widehat{\mathcal{P}_f}$, they follow from accessibility and mono-preservation of $\mathcal{F}$, $\mathcal{P}_c$ and $\mathcal{P}_f$ (see Proposition 3.1.19), as (co)limits in presheaf categories are computed pointwise. For $\mathcal{K}$ and $\mathcal{U}$, these properties are proven in [BZ15]. □

Therefore the terminal sequence for $\mathsf{At} \times \mathcal{S}(\cdot)$ converges at some limit ordinal, say γ, yielding the final $\mathsf{At} \times \mathcal{S}(\cdot)$-coalgebra $X_\gamma \xrightarrow{\simeq} At \times \mathcal{S}(X_\gamma)$. Then we can canonically define the final coalgebra semantics.

Definition 3.2.11. The *derivation semantics for general PLP* is defined as $[\![-]\!]_{p^\sharp} \colon \mathsf{At} \to X_\gamma$ by the universal property of the final coalgebra X_γ, shown in the diagram below.

$$\begin{tikzcd}
At \arrow[r, dotted, "{[\![-]\!]_{\mathsf{p}^\sharp}}"] \arrow[d, "{<id_{\mathsf{At}}, \mathsf{p}^\sharp>}"'] & X_\gamma \arrow[d, "\simeq"] \\
At \times \mathcal{S}(At) \arrow[r, "{id_{\mathsf{At}} \times [\![-]\!]_{\mathsf{p}^\sharp}}"'] & \mathsf{At} \times \mathcal{S}(X_\gamma)
\end{tikzcd} \tag{3.19}$$

A careful inspection of the terminal sequence constructing X_γ allows to infer a representation of its elements as trees, among which we have those representing computations by unification of goals in a PLP program. We call these *stochastic saturated derivation trees*, as they extend the derivation trees of Definition 3.1.11 and are the probabilistic variant of saturated and-or trees in [BZ15]. Using (3.19) one can easily verify that $[\![A]\!]_{\mathsf{p}^\sharp}$ is indeed the stochastic saturated derivation tree for a given goal A. Example 3.2.2 provides a pictorial representation of one such tree.

Definition 3.2.12 (Stochastic saturated derivation trees)**.** Given a probabilistic logic program $\mathbb{P}$, a natural number n and an atom $A \in \mathsf{At}(n)$. The *stochastic saturated derivation tree* for A in $\mathbb{P}$ is the possibly infinite tree $\mathcal{T}$ satisfying the following properties:

1. There are four kinds of nodes: atom-node (labelled with an atom), substitution-node (labelled with a substitution), clause-node (labelled with •), instance-node (labelled with ⧫), appearing alternatively in depth in this order.In particular, the root is an atom-node with label A.

2. Each edge between a substitution node and its clause-node children is labelled with some $r \in [0,1]$.

3. Suppose an atom-node s is labelled with $A' \in \mathsf{At}(n')$. For every substitution $\theta \colon n' \to m'$, s has exactly one (substitution-node) child t labelled with θ. For every clause $r :: H \leftarrow A_1, \ldots, A_k$ in $\mathbb{P}$ such that H matches $A'\theta$ (via some substitution), t has exactly one (clause-)child u, and edge $t \to u$ is labelled with r. Then for every substitution τ such

that $A'\theta = H\tau$ and $B_1\tau, \dots, B_k\tau \in \mathsf{At}(m')$, u has exactly one (instance-)child v. Also v has exactly $|\{B_1, \dots, B_k\}|\tau$-many (atom-)children, each labelled with one element in $\{B_1, \dots, B_k\}\tau$.

Proposition 3.2.13. *The stochastic derivation tree of an atom A in context n is $[\![A]\!]_{p^\sharp}(n)$, up to isomorphism.*

Proof. The proof is similar to that for the derivation trees of general logic programs in Bonchi and Zanasi [BZ13], where the only difference is the probability labels here from substitution nodes to clause nodes. □

Sometimes when the context n is clear, we are sloppy and simply write $[\![A]\!]_{p^\sharp}$ for $[\![A]\!]_{p^\sharp}(n)$. Intuitively, the stochastic derivation tree of A encodes the proof-search of A in $\mathbb{P}$, using unification. The probability labels encode the possibility of the clauses used along the search.

3.2.3 Distribution Semantics

In this section we conclude by giving a coalgebraic perspective on the distribution semantics $\langle\!\langle - \rangle\!\rangle$ for general PLP. Mimicking the ground case (Section 3.1.3), this will be presented as an extension of the derivation semantics, via a 'possible worlds' natural transformation. Also in the general case, we want to guarantee that a single probability value is computable for a given goal A from the corresponding tree $\langle\!\langle A \rangle\!\rangle$ in the final coalgebra — whenever this probability is also computable in the 'traditional' way (see (2.1)) of giving distribution semantics to PLP. In this respect, the presence of variables and substitutions poses additional challenges, for which we refer to Section 3.2.4. In a nutshell, the issue is that the distribution semantics counts the existence of a clause in the program rather than the number of times a clause is used in the computation. This means that every clause is counted at most once, independently from how many times that clause is used again in the computation. Note that our tree representation, as in the ground case, presents the resolution (thus computation aspect) of the program. So in our tree representation we need to give enough information to determine which clause is used at each step of the computation, so that a second use can be easily detected. However, neither our saturated derivation trees, nor a 'naive' extension of them to distribution trees, carry such information: what appears in there is only the instantiated heads and bodies, but in general one cannot retrieve the atom A from a substitution θ and the instantiation $A\theta$. This is best illustrated via a simple example.

Example 3.2.14. Consider the following program, based on the signature $\Sigma = \{a^0\}$ and two 1-ary predicates P, Q. It consists of two clauses:

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

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

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

$$\tilde{\mathsf{p}} \colon \mathcal{U}\mathsf{At} \to \widehat{\mathcal{M}_{pr}}(\widehat{\mathcal{P}_c}\widehat{\mathcal{P}_f}\mathcal{U}\mathsf{At} \times (\mathcal{U}\mathsf{At} \times \widehat{\mathcal{P}_f}\mathcal{U}\mathsf{At})) \tag{3.20}$$

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

$$\tilde{\mathsf{p}}_n(A) \colon \langle \{B_1\tau, \ldots, B_k\tau\}_{\tau \in \Gamma \subseteq \mathsf{mor}(\mathcal{L}_\Sigma^{op})}, \langle H, \{B_1, \ldots, B_k\}\rangle\rangle \mapsto \begin{cases} r & (r :: H \leftarrow B_1, \ldots, B_k) \in \mathbb{P} \\ & \text{ and } H\tau_i = A \\ 0 & \text{otherwise} \end{cases}$$

As in the case of p in (3.17), we can move from term-matching to unification by using the universal property of the adjunction $\mathcal{U} \dashv \mathcal{K}$, yielding $\tilde{\mathsf{p}}^\sharp \colon \mathsf{At} \to \mathcal{K}\widehat{\mathcal{M}_{pr}}(\widehat{\mathcal{P}_c}\widehat{\mathcal{P}_f}\mathcal{U}\mathsf{At} \times (\mathcal{U}\mathsf{At} \times \widehat{\mathcal{P}_f}\mathcal{U}\mathsf{At}))$. For simplicity henceforth we denote the functor $\mathcal{K}\widehat{\mathcal{M}_{pr}}(\widehat{\mathcal{P}_c}\widehat{\mathcal{P}_f}\mathcal{U}(\cdot) \times (\mathcal{U}\mathsf{At} \times \widehat{\mathcal{P}_f}\mathcal{U}\mathsf{At}))$ by $\mathcal{R}$.

As noted in Remark 3.2.8, instantiating $\tilde{\mathsf{p}}$ to some $n \in \mathsf{ob}(\mathcal{L}_\Sigma^{op})$ fixes a variable context $\{x_1, \ldots, x_n\}$ both for the goal and the clause labels. In practice, because the set of clauses is always finite, it suffices to chose n 'big enough' so that the variables appearing in the clauses are included in $\{x_1, \ldots, x_n\}$.

We are now able to conclude our characterisation of the distribution semantics. The 'possible worlds' transformation $pw \colon \mathcal{M}_{pr} \Rightarrow \mathcal{D}_{\leq}\mathcal{P}_f$ (Definition 3.1.31) yields a natural transformation $\widehat{pw} \colon \widehat{\mathcal{M}_{pr}} \to \widehat{\mathcal{D}_{\leq}\mathcal{P}_f}$, defined pointwise by pw. We can use $\widehat{pw}$ to translate $\mathcal{R}$ into the functor $\mathcal{K}\widehat{\mathcal{D}_{\leq}\mathcal{P}_f}(\widehat{\mathcal{P}_c}\widehat{\mathcal{P}_f}\mathcal{U}(\cdot) \times (\mathcal{U}\mathsf{At} \times \widehat{\mathcal{P}_f}\mathcal{U}\mathsf{At}))$, abbreviated as $\mathcal{O}$, which is going to give the type of saturated distribution trees for general PLP programs.

As a simple extension of the developments in Section 3.2.2, we can construct the cofree $\mathcal{R}$-coalgebra $\Phi \xrightarrow{\cong} \mathsf{At} \times \mathcal{R}(\Phi)$ via a terminal sequence. Similarly, one can obtain the final $\mathsf{At} \times \mathcal{O}$-coalgebra, or equivalently the cofree $\mathcal{O}$-coalgebra at At, which we denote as $\Psi \xrightarrow{\cong} \mathsf{At} \times \mathcal{O}(\Psi)$. By the universal property of Ψ, all these ingredients get together in the definition of the distribution semantics $\langle\!\langle - \rangle\!\rangle_{\tilde{\mathsf{p}}^\sharp}$ for arbitrary PLP programs $\tilde{\mathsf{p}}^\sharp$.

Definition 3.2.15. The *coalgebraic distribution semantics* $\langle\!\langle - \rangle\!\rangle_{\tilde{\mathsf{p}}^\sharp}$ is defined as follows using the universal mapping property of the final $\mathsf{At} \times \mathcal{O}$-coalgebra $\Psi \xrightarrow{\simeq} \mathsf{At} \times \mathcal{O}(\Psi)$.

$$\begin{tikzcd}
\mathsf{At} \arrow[r, dashed, "!_\Phi"] \arrow[rr, dashed, bend left, "\langle\!\langle - \rangle\!\rangle_{\tilde{\mathsf{p}}^\sharp}"] \arrow[d, "<id_{At}{,}\tilde{\mathsf{p}}^\sharp>"] & \Phi \arrow[r, dashed, "!_\Psi"] \arrow[d, "\cong"] & \Psi \arrow[dd, "\cong"] \\
\mathsf{At} \times \mathcal{R}At \arrow[r, "id_{At} \times \mathcal{R}(!_\Phi)"] \arrow[d, "id_{\mathsf{At}} \times \mathcal{K}\widehat{pw}"] & At \times \mathcal{R}\Phi \arrow[d, "id_{\mathsf{At}} \times \mathcal{K}\widehat{pw}"] & \\
\mathsf{At} \times \mathcal{O}\mathsf{At} \arrow[r, "id_{\mathsf{At}} \times \mathcal{O}(!_\Phi)"] & \mathsf{At} \times \mathcal{O}\Phi \arrow[r, "id_{\mathsf{At}} \times \mathcal{O}(!_\Psi)"] & \mathsf{At} \times \mathcal{O}\Psi
\end{tikzcd}$$

where $!_\Phi$ and $!_\Psi$ are given by the evident universal properties, and show the role of the cofree $\mathcal{R}$-coalgebra Φ as an intermediate step.

The layered construction of the final coalgebras Ψ and Φ, together with the above characterisation of $\langle\!\langle - \rangle\!\rangle_{\tilde{\mathsf{p}}^\sharp}$, allow to conclude that the distribution semantics for the program $\tilde{\mathsf{p}}^\sharp$ maps a goal A to its *saturated distribution tree* $\langle\!\langle A \rangle\!\rangle_{\tilde{\mathsf{p}}^\sharp}$, as formally defined below.

Definition 3.2.16 (Saturated distribution tree)**.** The *saturated distribution tree* for $A \in \mathsf{At}(n)$ in $\mathbb{P}$ is the possibly infinite $\mathcal{T}$ satisfying the following properties based on Definition 3.2.12:

1. There are five kinds of nodes: in addition to the atom-, substitution-, clause- and instance-nodes, there are world-nodes. The world-nodes are children of the substitution-nodes, and parents of the clause nodes. The root and the order of the rest of the nodes are the same as in Definition 3.2.12, condition **1**. The clause-nodes are now labelled with clauses in $\mathbb{P}$.

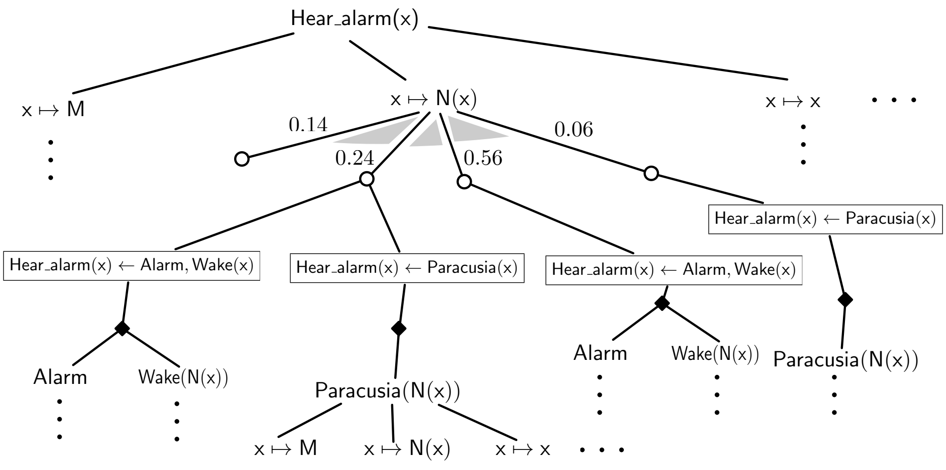
2. Suppose s is an atom-node labelled with $A' \in \mathsf{At}(n')$, and t is a substitution-child of s labelled with $\theta \colon n' \to m$. Let C be the set of all clauses φ such that $\mathsf{head}(\varphi)$ matches $A'\theta$. Then t has $2^{|C|}$ world-children, each representing a subset X of C. If a world-child u of t represents subset X, then the edge $t \to u$ has probability label $\prod_{\varphi \in X} lab(\varphi) \cdot \prod_{\varphi' \in C \setminus X}(1 - lab(\varphi'))$. Also u has $|X|$ clause-children, one for each clause $\varphi \in X$, labelled with the corresponding clause. The rest for clause-nodes and instance-nodes are the same as in Definition 3.2.12, condition **3**.

Proposition 3.2.17. *The saturated distribution tree for atom A in context n is $\langle\!\langle A \rangle\!\rangle_{\tilde{\mathsf{p}}^\sharp}(n)$.*

Remark 3.2.18. Note that, in principle, saturated distribution trees could be defined coalgebraically without the intermediate step of adding clause labels. This is to be expected: coalgebra typically captures the one-step, 'local' behaviour of a system. On the other hand, as explained, the need for clause labels is dictated by a computational aspect involving the depth of distribution trees, that is, a 'non-local' dimension of the system.

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

Example 3.2.19. In the context of Example 3.2.1, the tree $\langle\!\langle \texttt{Hear\_alarm(x)} \rangle\!\rangle$ capturing the distribution semantics of $\texttt{Hear\_alarm(x)}$ is (partially) depicted as follows. Note the presence of clauses labelling the clause-nodes.



3.2.4 Computability of the distribution semantics

In this subsection we slightly turn away from coalgebras and justify the tree representation of the distribution semantics established so far. Before we claimed that the probability $\Pr_{\mathbb{P}}(A)$ associated with a goal (see (2.1)) can be straightforwardly computed from the corresponding distribution tree $\langle\!\langle A \rangle\!\rangle_p$, and here supplies such algorithms, for both the ground case and general case. Note that this serves just as a proof of concept, without any claim of efficiency compared to pre-existing implementations, algorithms using BDD in e.g. De Raedt et al. [DRKT07a].

3.2.4.1 Ground case PLP.

We start with the simpler case of ground programs. Let us fix a ground PLP program $\mathbb{P}$ with atoms At, a goal $A \in At$ and the distribution tree $\mathcal{T}$ for A in $\mathbb{P}$ (see Definition 3.1.25).

First, we claim that without loss of generality we may assume that the distribution tree $\mathcal{T}$ is finite. Indeed, in the ground case, infinite branches only result from loops, namely multiple appearances of an atom in some path, which can be easily detected (this is no longer the case in the following general case). We can always prune the subtrees of $\mathcal{T}$ rooted by atoms that already appeared at an earlier stage: this does not affect the computation of $\Pr_{\mathbb{P}}(A)$, and it makes $\mathcal{T}$ finite.

Next, we introduce the concept of *deterministic subtrees* of distribution trees, which can be seen as the chose possible worlds. Basically a deterministic subtree of a distribution tree $\mathcal{T}$ can be obtained by traversing from the root, such that at each atom node one selects exactly one child and prune the rest. Moreover, this slection should be uniform for any two aomts nodes with the same atom label. Recall that every clause-node t in $\mathcal{T}$ represents a clause φ in the pure logic program $|\mathbb{P}|$, whose head $\mathsf{head}(\varphi)$ is the label of the atom-grandparent of t, and body $\mathsf{body}(\varphi)$ consists of the labels of the atom-children of t.

Definition 3.2.20. A subtree $\mathcal{S}$ of a distribution tree $\mathcal{T}$ is *deterministic* if (i) it contains exactly one child (world-node) for each atom-node and all children for other nodes, and (ii) for any distinct atom-nodes s, t in $\mathcal{S}$ with the same label, s and t have their clause-grandchildren representing the same clauses.

The idea is that $\mathcal{S}$ describes a computation in which the choice of a possible world (*i.e.* a sub-program of $\mathbb{P}$) associated to any atom B appearing during the resolution is uniquely determined. Because of this feature, each deterministic subtree uniquely identifies a set of sub-programs of $\mathbb{P}$, and together the deterministic subtrees of $\mathcal{T}$ form a *partition* over the set of these sub-programs (see Proposition 3.2.23 below).

Example 3.2.21. Consider the following program $\mathbb{P}_{deter}$:

$$\begin{array}{rlll@{\qquad}rlll} 0.5 :: & A & \leftarrow B, C & & 0.5 :: & C & \leftarrow E \\ 0.5 :: & B & \leftarrow C & & 1 :: & D & \leftarrow \\ 0.5 :: & C & \leftarrow D & & 1 :: & E & \leftarrow \end{array}$$

Then for the following two subtrees of the distribution tree of A, the one on the right is deterministic, while the one on the left is not deterministic. In the latter case, it does not represent a sub-program: at the two atom-nodes labelled with C, two different world nodes are chosen, one contains only the clause $C \leftarrow D$, and the other only the clause $C \leftarrow E$.

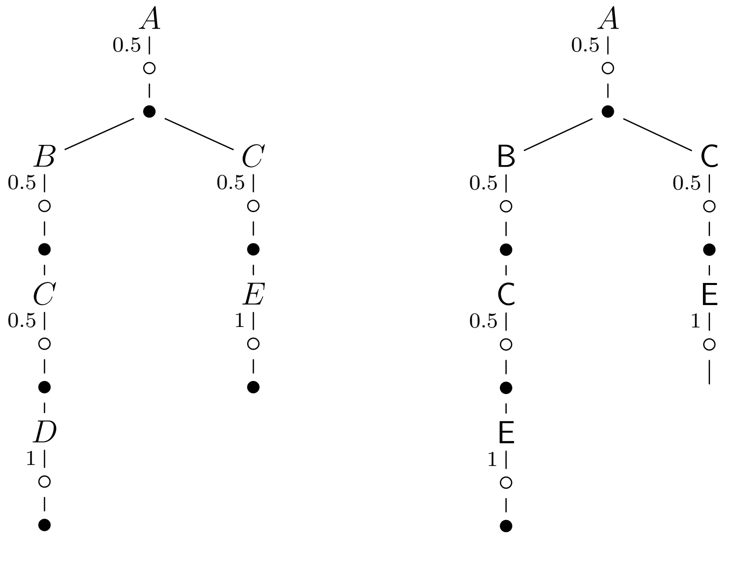


Figure 3.1: (Non-)deterministic subtrees in $\mathbb{P}_{deter}$

Since $\mathcal{T}$ is finite, it is clear that we can always provide an enumeration of its deterministic subtrees. We can now present our algorithm, in two steps (see Figure 3.2). First, Algorithm 1 computes the probability associated with a deterministic subtree. Second, Algorithm 2 computes $\Pr_{\mathbb{P}}(A)$ by summing up the probabilities found by Algorithm 1 on all the deterministic

Algorithm 1 Compute probability of a deterministic subtree

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

prob_list = [ ]
for atom-node $s$ in $\mathcal{S}$ do
    if $s$ has child then
        prob_list.append(label($s \to$ child($s$)))
if prob_list == [ ] then
    return 0
else prob = product of values in prob_list
    return prob
```

Algorithm 2 Compute probability of a goal

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

prob_suc = 0
for deterministic subtree $\mathcal{S}$ of $\mathcal{T}$ do
    if $\mathcal{S}$ proves $A$ then
        prob_suc += Algorithm 1($\mathcal{S}$)
return prob_suc
```

Figure 3.2: Compute distribution semantics from distribution trees, ground case

subtrees of $\mathcal{T}$ that contains a refutation of A. In the algorithms we use label($s \to t$) to denote the probability value labelling the edge from s to t.

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

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

Definition 3.2.22. Given a clause φ in $\mathbb{P}$, a deterministic subtree $\mathcal{S}$ of $\mathcal{T}$, a world-node t, and its atom-parent s in $\mathcal{S}$, we say node t *accepts* clause φ if $\mathsf{head}(\varphi) = \text{label}(s)$ and there is a clause-child of t that represents φ; node t *rejects* clause φ if $\mathsf{head}(\varphi) = \text{label}(s)$ but no clause-child of t represents φ. We say the deterministic subtree $\mathcal{S}$ *accepts* (*rejects*) φ if there exists a world-node t in $\mathcal{S}$ that accepts (rejects) φ.

Note that in Definition 3.2.20, condition (ii) prevents the existence of world-nodes t, t' in $\mathcal{S}$ such that t accepts φ and t' rejects φ. Thus the notion that $\mathcal{S}$ accepts (rejects) φ is well-defined.

We denote the set of clauses accepted and rejected by $\mathcal{S}$ by $\mathsf{Acc}(\mathcal{S})$ and $\mathsf{Rej}(\mathcal{S})$, respectively. Then we can define the set $\mathit{SubProg}(\mathcal{S})$ of sub-programs represented by $\mathcal{S}$ as

$$\mathit{SubProg}(\mathcal{S}) \coloneqq \{\mathbb{L} \subseteq |\mathbb{P}| \mid \forall\varphi \in \mathsf{Acc}(\mathcal{S}), \varphi \in \mathbb{L}; \forall\psi \in \mathsf{Rej}(\mathcal{S}), \psi \notin \mathbb{L}\} \tag{3.21}$$

The correctness of the algorithms are then proven through the following basic observations on the connection between deterministic subtrees and the sub-programs they represent:

Proposition 3.2.23. *Suppose $\mathcal{S}$ is a deterministic subtree of the distribution tree $\mathcal{T}$ of A.*

1. *$\{\mathit{SubProg}(\mathcal{S}) \mid \mathcal{S}$ is deterministic subtree of $\mathcal{T}\}$ forms a partition of $\mathcal{P}(|\mathbb{P}|)$.*

2. *Either $\mathbb{L} \vdash A$ for all $\mathbb{L} \in \mathit{SubProg}(\mathcal{S})$ or $\mathbb{L} \nvdash A$ for all $\mathbb{L} \in \mathit{SubProg}(\mathcal{S})$.*

3. *$\sum_{\mathbb{L}\in \mathit{SubProg}(\mathcal{S})} \Pr_{\mathbb{P}}(\mathbb{L}) = \prod_{r_i\in\mathcal{S}} r_i$, where the r_is are all the probability labels appearing in $\mathcal{S}$ (on the atom-node → world-node edges).*

Proof. 1. Given any two distinct deterministic subtrees, there is an atom-node s in $\mathcal{T}$ such that the subtrees include distinct world-child of s. So by (3.21) the sub-programs they represent differ at least regarding one clause. Moreover, given a sub-program $\mathbb{L}$, one can always identify a deterministic subtree $\mathcal{S}$ such that $\mathbb{L} \in \mathit{SubProg}(\mathcal{S})$, as follows: given the A-labelled root of $\mathcal{T}$, select the world-child w of A representing the (possibly empty) set X of all clauses in $\mathbb{L}$ whose head is A; then select the children (if any) of w, and repeat the procedure.

2. Note that a sub-program $\mathbb{L} \in \mathit{SubProg}(\mathcal{S})$ proves the goal A iff $\mathcal{S}$ contains a successful refutation of A, and the latter property is independent of the choice of $\mathbb{L}$.

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

$$\mu_{\mathbb{P}}(\mathbb{L}) = \left(\prod_{\varphi\in\mathsf{Acc}(\mathcal{S})} \mathit{lab}(\varphi)\right)\cdot\left(\prod_{\varphi'\in\mathsf{Rej}(\mathcal{S})} (1-\mathit{lab}(\varphi'))\right)\cdot\mu_{\mathbb{P}\setminus(\mathsf{Acc}(\mathcal{S})\cup\mathsf{Rej}(\mathcal{S}))}(\mathbb{L}\setminus\mathsf{Acc}(\mathcal{S})) \tag{3.22}$$

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

$$\sum_{\mathbb{L}\in\mathit{SubProg}(\mathcal{S})} \mu_{\mathbb{P}\setminus(\mathsf{Acc}(\mathcal{S})\cup\mathsf{Rej}(\mathcal{S}))}(\mathbb{L}\setminus\mathsf{Acc}(\mathcal{S})) = 1. \tag{3.23}$$

Applying equation (3.23) to the sum of (3.22) over all $\mathbb{L} \in \mathit{SubProg}(\mathcal{S})$, we get

$$\sum_{\mathbb{L}\in\mathit{SubProg}(\mathcal{S})} \mu_{\mathbb{P}}(\mathbb{L}) = \prod_{\varphi\in\mathsf{Acc}(\mathcal{S})} \mathit{lab}(\varphi)\cdot\prod_{\varphi'\in\mathsf{Rej}(\mathcal{S})} (1-\mathit{lab}(\varphi')) \tag{3.24}$$

For each world-node t and its atom-parent s, we can use the terminology in Definition 3.2.22, and express label$(s \to t)$ (see Definition 3.1.25) as

$$\text{label}(s \to t) = \prod_{t \text{ accepts } \varphi} lab(\varphi) \cdot \prod_{t \text{ rejects } \varphi'} (1 - lab(\varphi')). \tag{3.25}$$

Applying (3.25) to the whole deterministic subtree $\mathcal{S}$, we obtain

$$\begin{aligned}\sum_{\mathbb{L} \in SubProg(\mathcal{S})} \mu_{\mathbb{P}}(\mathbb{L}) &\overset{(3.24)}{=} \prod_{\varphi \in \mathsf{Acc}(\mathcal{S})} lab(\varphi) \cdot \prod_{\varphi' \in \mathsf{Rej}(\mathcal{S})} (1 - lab(\varphi')) \\ &\overset{\text{Def.3.2.22}}{=} \prod_{(\text{world-node } t \text{ in } \mathcal{S})} \left(\prod_{t \text{ accepts } \varphi} lab(\varphi) \cdot \prod_{t \text{ rejects } \varphi'} (1 - lab(\varphi')) \right) \\ &\overset{(3.25)}{=} \prod_{r_i \in \mathcal{S}} r_i\end{aligned}$$

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

We can now formulate the success probability of A as follows

$$\begin{aligned}Pr_{\mathbb{P}}(A) = \sum_{|\mathbb{P}| \supseteq \mathbb{L} \vdash A} \mu_{\mathbb{P}}(\mathbb{L}) &\overset{(\text{Prop.3.2.23,1\&2})}{=} \sum_{\mathcal{S} \vdash A} \sum_{\mathbb{L} \in SubProg(\mathcal{S})} \mu_{\mathbb{P}}(\mathbb{L}) \\ \overset{(3.24)}{=} \sum_{\mathcal{S} \vdash A} \left(\prod_{\varphi \in \mathsf{Acc}(\mathcal{S})} lab(\varphi) \cdot \prod_{\varphi' \in \mathsf{Rej}(\mathcal{S})} (1 - lab(\varphi')) \right) &\overset{(\text{Prop.3.2.23,3})}{=} \sum_{\mathcal{S} \vdash A} \prod_{r_i \in S} r_i\end{aligned}$$

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

3.2.4.2 General case PLP

Computability of the distribution semantics for arbitrary PLP programs relies on the substitution mechanism employed in the resolution. This aspect deserves a preliminary discussion. Traditionally, logic programming has both the theorem-proving and problem-solving perspectives [KP18]. From the problem-solving perspective, the aim is to find a proof of *some substitution instance* of G. From the theorem-proving perspective, the aim is to search for a proof of the goal G itself as an atom. The main difference is in the substitution mechanism of resolution: one uses unification for the problem-solving and term-matching for the theorem-proving perspective.

We will first explore computability within the theorem-proving perspective. As resolution therein is by term-matching, the probability $Pr^{TM}_{\mathbb{P}}(A)$ of proving a goal A in a PLP program $\mathbb{P}$ is formulated as $Pr^{TM}_{\mathbb{P}}(A) \coloneqq \sum_{|\mathbb{P}| \supseteq \mathbb{L} \vdash A} \mu_{\mathbb{P}}(\mathbb{L})$.

In our coalgebraic framework, the distribution semantics for general PLP programs is represented on 'saturated' trees, in which computations are performed by unification. However, following [BZ15], one can define the *TM (**T**erm **M**atching) distribution tree* of a goal A in a program $\mathbb{P}$ by 'desaturation' of the saturated distribution tree for A in $\mathbb{P}$. The coalgebraic definition, for which we refer to [BZ15], applies pointwise on the saturated tree the counit $\epsilon_{\mathcal{U}At} \colon \mathcal{U}\mathcal{K}\mathcal{U}\mathsf{At} \to \mathcal{U}\mathsf{At}$ of the adjunction $\mathcal{U} \dashv \mathcal{K}$ (*cf.* (3.16)), whose details we spell out here. The counit $\varepsilon \colon \mathcal{U} \circ \mathcal{K} \Rightarrow \mathbf{1}_{\mathsf{Set}^{|\mathcal{L}_{\Sigma}^{op}|}}$ of the adjunction $\mathcal{U} \dashv \mathcal{K}$ is defined on a presheaf $\mathcal{F} \colon |\mathcal{L}_{\Sigma}^{op}| \to \mathsf{Set}$ as follows: for arbitrary $n \in \mathsf{ob}(|\mathcal{L}_{\Sigma}^{op}|)$ and $\dot{x} \in \mathcal{U}\mathcal{K}(\mathcal{F})(n)$,

$$\varepsilon_{\mathcal{F}}(n) \colon \dot{x} \mapsto \dot{x}(id_n)$$

Recall that $\dot{x} \in \mathcal{U}\mathcal{K}(\mathcal{F})(n)$ is a tuple in $\prod_{\theta \in \mathcal{L}_{\Sigma}^{op}[n,m]} \mathcal{F}(m)$, so $\varepsilon_{\mathcal{F}}(n)(\dot{x})$ is the component of $\dot{x}$ at index id_n.

The TM distribution tree which results from 'desaturation' can be described very simply: at each layer of the starting saturated distribution tree, one prunes all the subtrees which are not labelled with the identity substitution $id := x_1 \mapsto x_1, x_2 \mapsto x_2, \dots$. In this way, the only remaining computation are those in which resolution only applies a non-trivial substitution on the clause side, that is, in which unification is restricted to term-matching.

We formalise the idea above using a construction on the terminal sequences. First, we identify TM distribution trees as elements in certain final coalgebra. The coalgebra type is determined by the form of TM distribution trees, namely $\mathcal{U}\mathsf{At} \times \widehat{\mathcal{D}_{\leq}}\widehat{\mathcal{P}_f}(\widehat{\mathcal{P}_c}\widehat{\mathcal{P}_f}(-) \times (\mathcal{U}\mathsf{At} \times \widehat{\mathcal{P}_f}\mathcal{U}\mathsf{At})) \colon \mathsf{Set}^{|\mathcal{L}_{\Sigma}^{op}|} \to \mathsf{Set}^{|\mathcal{L}_{\Sigma}^{op}|}$, which we denote as $\mathcal{U}\mathsf{At} \times \mathcal{Q}$.

Construction 3.2.24. The terminal sequence for $\mathcal{U}\mathsf{At} \times \mathcal{Q}$ consists of a sequence of objects $\{Z_\alpha\}_{\alpha \in Ord}$ and morphisms $\zeta_\alpha^\beta \colon Z_\beta \to Z_\alpha$ for $\alpha \leq \beta$ defined by induction on α in the same pattern as Construction 3.1.28. For instance,

$$Z_\alpha = \begin{cases} \mathcal{U}\mathsf{At} & \alpha = 0 \\ \mathcal{U}\mathsf{At} \times \mathcal{Q}(X_\beta) & \alpha = \beta + 1 \\ \text{the expected limit} & \alpha \text{ is limit ordinal} \end{cases}$$

Since $\mathcal{Q}$ is a mono-preserving accessible functor, then the sequence converges at some limit ordinal χ such that $Z_\chi \cong Z_{\chi+1}$. Then this Z_χ is the final $\mathcal{U}\mathsf{At} \times \mathcal{Q}$-coalgebra, or equivalently, the cofree coalgebra of $\mathcal{Q}$ at $\mathcal{U}\mathsf{At}$, namely $\mathcal{C}(\mathcal{Q})(\mathcal{U}\mathsf{At})$.

Then the aim is to construct a morphism $d \colon \mathcal{U}(\mathcal{C}(\mathcal{O})(\mathsf{At})) \to \mathcal{C}(\mathcal{Q})(\mathcal{U}\mathsf{At})$: since every saturated distribution tree is an element in $\mathcal{C}(\mathcal{O})(\mathsf{At})$ thus in $\mathcal{U}(\mathcal{C}(\mathcal{O})(\mathsf{At}))$, a proper morphism of the above type maps a distribution tree to its TM distribution tree. Intuitively, such d performs desaturation via $\varepsilon^{\mathcal{K}\mathcal{U}}$ layer by layer to the saturated distribution tree. The method is to construct a sequence of morphisms between the terminal sequence, whose universal mapping property induces the desired d. Let us suppose that the terminal sequence of $\mathcal{O}$ is $\{X_\alpha\}_{\alpha \in Ord}$

and $\{\delta^{\beta}_{\alpha}\}_{\alpha\leq\beta}$, and it terminates at some limit ordinal γ.

Construction 3.2.25. Consider the terminal sequence $\{\delta^{\beta}_{\alpha}\colon X_\beta \to X_\alpha\}$ for $\mathcal{O}$ and the terminal sequence $\{\zeta^{\beta}_{\alpha}\colon Z_\beta \to Z_\alpha\}$ for $\mathcal{Q}$ (Construction 3.2.24). We define a sequence of natural transformations $\{d_\alpha\colon \mathcal{U}X_\alpha \to Z_\alpha\}_{\alpha\in Ord}$ inductively as follows:

- For $\alpha = 0$, $d_0 = id_{\mathcal{U}\mathsf{At}}$.

- For successor ordinal $\alpha = \beta+1$, $d_\alpha = id_{\mathcal{U}\mathsf{At}} \times (\mathcal{Q}(d_\beta) \circ \varepsilon^{\mathcal{U}\mathcal{K}}_{\mathcal{O}^{-}X_\beta})$, where $\mathcal{O}^{-}$ is the functor obtained by removing the $\mathcal{K}$ in $\mathcal{O}$.

- For limit ordinal α, $d_\alpha\colon \mathcal{U}X_\alpha \to Z_\alpha$ is defined using the universal mapping property of Z_α, since the latter is defined as the limit of the terminal sequence $\{\zeta^{\alpha}_{\xi}\}_{\xi<\alpha}$ preceding Z_α.

Now we are ready to define $d\colon \mathcal{U}(\mathcal{C}(\mathcal{O})(\mathsf{At})) \to \mathcal{C}(\mathcal{Q})(\mathcal{U}\mathsf{At})$. Equivalently, using the terminal sequence construction such d is of the form $\mathcal{U}X_\gamma \to Z_\chi$, so we make a case distinction on $\gamma \leq \chi$ and $\gamma > \chi$ (note that there is no guarantee that these two terminal sequences converge at the same ordinal). If $\gamma \leq \chi$, then $X_\chi \cong X_\gamma$, and let d simply be d_χ. If $\gamma > \chi$, then $Z_\gamma \cong Z_\chi$, and let d be $d_\gamma \circ \delta^{\gamma}_{\chi}$.

Proposition 3.2.26. *Given arbitrary $n \in \mathsf{ob}(\mathcal{L}^{op}_{\Sigma})$ and $A \in \mathsf{At}(n)$, the TM distribution tree of A in $\mathbb{P}$ is $\left(d \circ \mathcal{U}\left(\langle\!\langle - \rangle\!\rangle_{\tilde{\mathsf{p}}^\sharp}\right)\right)(n)(A)$, where $\langle\!\langle - \rangle\!\rangle_{\tilde{\mathsf{p}}^\sharp}$ is defined in Definition 3.2.15.*

So far we have formally shown how, in our coalgebraic picture, one can obtain the TM distribution tree from the saturated one. We conclude this part with a few remarks on how to compute the probability of atoms in general PLP, given the algorithms for ground PLP.

For the term-matching perspective, one may compute the success probability $Pr^{TM}_{\mathbb{P}}(A)$ from the TM distribution tree of A. The computation goes similarly to Algorithm 2 : the problem amounts to calculating the probabilities of those deterministic subtrees of the distribution tree which prove the goal. However there are some crucial difference compared to the ground case.

1. The probability $Pr^{TM}_{\mathbb{P}}(A)$ is not computable in whole generality. It depends on whether one can decide all the proofs of A in the pure logic program $|\mathbb{P}|$, and there are various heuristics in logic programming for this task.

2. It is still possible to decide whether a subtree is deterministic, but the algorithm in the general case is a bit subtler, as it is now possible that two different goals match the same clause (instantiated in two different ways).

3. When calculating the probability of a deterministic subtree in the TM distribution tree, multiple appearances of a single clause (possibly instantiated with different substitutions) should be counted only once. In order to ensure this one needs to be able to identify which clause is applied at each step of the computation described by the distribution trees: this is precisely the reason of the addition of the clause labels in the coalgebra type of these trees, as discussed in Section 3.2.3.

For the problem-solving perspective, where resolution is based on arbitrary unification rather than just term-matching, one works directly with the saturated distribution trees. In standard SLD-resolution, computability relies on the possibility of identifying the *most general* unifier between a goal and the head of a given clause. This can be done also within saturated distribution trees, since saturation supplies *all* the unifiers, thus in particular the most general one. This means that, in principle, one may compute the distribution semantics based on most general unification from the saturated distributed tree associated with a goal, with similar caveats as the ones we described for the term-matching case. However, the lack of a satisfactory coalgebraic treatment of most general unifiers [BZ15] makes us prefer the theorem-proving perspective discussed above, for which desaturation provides an elegant categorical formalisation.

3.3 Coalgebraic perspective of WLP

In this part we will present the coalgebraic semantics for WLP analogous to that for PLP in Section 3.1. We focus on ground WLP, where a coalgebraic presentation of WLP programs is given, followed by a derivation semantics (Section 3.3.1). The computation of the weight semantics from the derivation semantics is in Section 3.3.2.

3.3.1 Coalgebraic semantics of ground WLP

Let us fix a finite set of atoms At, a semiring $\mathbf{K} = \langle K, +, \cdot, \mathbf{0}, \mathbf{1}\rangle$, and a ground WLP program $\mathbb{W}$ based on At, whose weights take value in $\mathbf{K}$. Then $\mathbb{W}$ can be represented as a coalgebra of the type $\mathcal{M}_{\mathbf{K}}\mathcal{P}_f \colon \mathsf{Set} \to \mathsf{Set}$, where $\mathcal{M}_{\mathbf{K}}$ is the multiset functor corresponding to the underlying semiring (see Section 2.1.3).

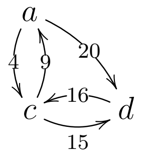
Definition 3.3.1. The WLP program $\mathbb{W}$ induces a $\mathcal{M}_{\mathbf{K}}\mathcal{P}_f$-coalgebra $w \colon At \to \mathcal{M}_{\mathbf{K}}\mathcal{P}_f(At)$ as follows. For each $A \in At$,

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

Example 3.3.2. Consider the following ground program $\mathbb{P}_{gsp}$ obtained as a simple ground version of $\mathbb{P}_{sp}$. The underlying semiring is still **MinPlus** $= \langle \mathbb{N} \cup \{+\infty\}, \min, +, +\infty, 0\rangle$.

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

The program describes the following (single-sourced, directed) weighted graph (3.26) where a is the entering node, such that the value of reachable(x) in the program represents the shortest path from the entering node to x:



(3.26)

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

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

This program induces a coalgebra $p^{gsp} \colon At_{gsp} \to \mathcal{M}_{\mathbf{MinPlus}}\mathcal{P}_f(At_{gsp})$, some of whose images are presented below:

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

Note that the 0 in the above equations is the $\cdot$-unit $\mathbf{1}$ in **MinPlus**, and should not be confused with the $+$-unit $\mathbf{0}$.

Unsurprisingly, there is a 1-1 correspondence between WLP programs over and $\mathcal{M}_{\mathbf{K}}$-coalgebras.

Proposition 3.3.3. *Let* $\mathbf{PropWLP_K}$ *be the category of propositional WLP programs based on* $\mathbf{K}$*, defined similar to* **PropPLP** *(Definition 3.1.7). Then* $\mathbf{PropWLP_K} \cong Fin(\mathbf{Coalg}(\mathcal{M}_{\mathbf{K}}\mathcal{P}_f))$.

Just as in the case of PLP, we may represent operationally the recursive calculation of the weight $weight(A)$ of a goal A as a certain kind of tree, which we call *weighted derivation tree.* These are weighted version of the and-or trees for pure logic programming [GC94]. They are formally defined analogously to the stochastic derivation trees in Definition 3.1.11, except that the probability labels are replaced by weight labels.

Definition 3.3.4. The *weighted derivation tree* for atom A in $\mathbb{W}$ is the possibly infinite tree satisfying the following conditions:

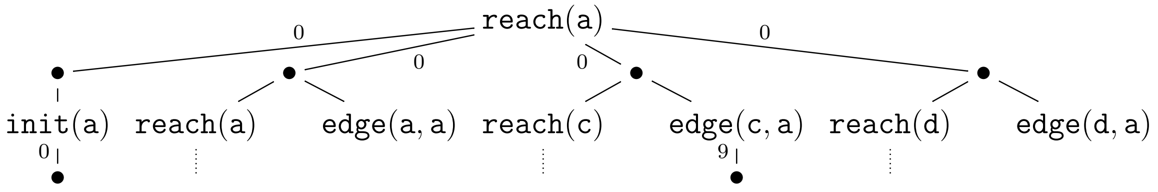
1. Every node is either an *atom-node* (labelled with an atom $B \in At$) or a *clause-node* (labelled with $\bullet$). These two types of nodes appear alternatingly in depth, in this order. In particular, the root is an atom-node labelled with A.

2. Each edge from an atom-node to its (clause-)children is labelled with a weight in $\mathbf{K}$.

3. Suppose s is an atom-node with label B. Then for every clause $r :: B \leftarrow B_1, \ldots, B_k$ in $\mathbb{P}$, s has exactly one child t such that the edge $s \to t$ is labelled with r, and t has exactly k children labelled with $B_1, \ldots, B_k$, respectively.

We illustrate the concept with an example.

Example 3.3.5. In the context of Example 3.3.2, the weighted derivation tree for $\mathsf{reachable}(\mathsf{a})$ is partially depicted below. Note the different meaning for an atom-node and a clause-node to have no child. For instance, the clause-node following `init(a)` has no child because '$0 ::$ `initial(a)` $\leftarrow$' is a clause in $\mathbb{P}_{gsp}$, while the atom-node $\mathtt{edge(d,a)}$ has no child because there is no clause in $\mathbb{P}_{gsp}$ whose head is $\mathtt{edge(d,a)}$.



(3.27)

By replacing $\mathcal{M}_{pr}$ by $\mathcal{M}_{\mathbf{K}}$ in Construction 3.1.17, one may construct the terminal sequence for the functor $At \times \mathcal{M}_{\mathbf{K}}\mathcal{P}_f(-)$, which by accessibility [Wor99] converges to a limit X_γ at some limit ordinal γ, yielding the final $At \times \mathcal{M}_{\mathbf{K}}\mathcal{P}_f$-coalgebra $X_\gamma \cong At \times \mathcal{M}_{\mathbf{K}}\mathcal{P}_f(X_\gamma)$, or equivalently, the cofree coalgebra $\mathcal{C}(\mathcal{M}_{\mathbf{K}}\mathcal{P}_f)(At)$. This gives us the final coalgebra semantics.

Definition 3.3.6. The coalgebraic semantics of WLP program $\mathbb{W}$ is the unique coalgebra morphism $[\![-]\!]_w \colon At \to \mathcal{C}(\mathcal{M}_{\mathbf{K}}\mathcal{P}_f)(At)$ induced by the universal property of the final $\mathcal{M}_{\mathbf{K}}\mathcal{P}_f$-coalgebra:

$$\begin{tikzcd}
At \arrow[r, dashed, "{[\![-]\!]_w}"] \arrow[d, "<id{,}w>"] & X_\gamma \arrow[d, "\simeq"] \\
At \times \mathcal{M}_{\mathbf{K}}\mathcal{P}_f(At) \arrow[r, "id\times\mathcal{M}_{\mathbf{K}}\mathcal{P}_f([\![-]\!]_w)"] & At \times \mathcal{M}_{\mathbf{K}}\mathcal{P}_f(X_\gamma).
\end{tikzcd}$$

By construction, one may readily verify that weighted derivation trees are elements of X_γ, and $[\![-]\!]_w$ maps an atom A to its weighted derivation tree, computed according to the program $\mathbb{W}$. Thus we can define the derivation semantics for WLP as the map $[\![-]\!]_w$.

Proposition 3.3.7. *The weighted derivation tree of A in $\mathbb{W}$ is $[\![A]\!]_w$.*

Note that, similar to that for logic programming and probabilistic logic programming, the weight $weight^w(A)$ of a goal is effectively computable from its semantics $[\![A]\!]_w$ — see Section 3.3.2 for details.

As a final remark, we choose not to explore the full details of the coalgebraic semantics of general WLP, mainly out of two reasons. First, to the best of our knowledge, most of the programs in WLP literature and application are either propositional or function-free, namely the signature of the language has no function symbol. As a consequence, given such a finite signature, its corresponding Herbrand space of the program is finite, as it only consists of the constants in the signature. Therefore one can always view a WLP clause with variables as the abbreviation of finitely many grounded clauses resulting from instantiating the clause with constants. For instance, recall the program $\mathbb{P}_{gsp}$ from Example 2.1.16. It has variables but is function-free, thus the clause $0 :: \texttt{reachable}(\texttt{x}) \leftarrow \texttt{reachable}(\texttt{y}), \texttt{edge}(\texttt{y}, \texttt{x})$ can be seen as the abbreviation of the ground clauses $0 :: \texttt{reachable}(\texttt{a}) \leftarrow \texttt{reachable}(\texttt{a}), \texttt{edge}(\texttt{a}, \texttt{a}).$, $0 :: \texttt{reachable}(\texttt{a}) \leftarrow \texttt{reachable}(\texttt{b}), \texttt{edge}(\texttt{b}, \texttt{a}), \dots, 0 :: \texttt{reachable}(\texttt{d}) \leftarrow \texttt{reachable}(\texttt{d}), \texttt{edge}(\texttt{d}, \texttt{d})$ in $\mathbb{P}_{sp}$ from Definition 3.3.1. These two programs have the same semantics for all ground atoms. Indeed, to obtain a well-defined semantics of weighted logic program in general (*i.e.* possible containing function symbols), one may need several extra assumption of the underlying semiring $\mathbf{K}$, such as $+$-complete and $\cdot$ distributes over countable $+$ sum. Second, even if one is to consider the coalgebraic representation of general WLP, there is minor adjustment given the saturated semantics for PLP in Section 3.2.4.2.

3.3.2 Computability of the weight semantics

To finalise the discussion on the coalgebraic semantics of WLP, we show how the weight $\mathit{weight}_{\mathbb{W}}(A)$ of an atom A in $\mathbb{W}$ is computable from its weighted derivation tree $[\![A]\!]_w$.

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

To make this precise in our computation, recall that a proof subtree for a goal A is a subtree of $[\![A]\!]_w$ where we select exactly one child for each atom-node with children. A proof tree is a successful one if it is finite and every leaf succeeds (i.e. is a clause node without child). We may define the weight of a finite proof subtree for A inductively as the weight of the root node u (labelled with A) in that particular tree using the following formula:

$$\mathit{weight}(u) = c \cdot \mathit{weight}(v_1) \cdot \cdots \cdot \mathit{weight}(v_k) \tag{3.28}$$

where c is the weight label of the (unique) edge from u, and $v_1, \dots, v_k$ are all the grandchildren (atom-nodes) of u. Note the overall weight of a goal A in $[\![A]\!]_w$ is then the $+$-sum of the weights of its proof subtrees ([CSS10]).

This means that, from a computational viewpoint, discarding a proof subtree is equivalent to assigning weight $\mathbf{0}$ (the $+$-unit) to that proof subtree. In order to assign weight $\mathbf{0}$ to each proof subtree with cycles, we look for the first appearance of a cycle, re-label the next edge with $\mathbf{0}$, and discard all the descendants. This works because, according to (3.28) and the fact that $\mathbf{0} \cdot c = c \cdot \mathbf{0} = \mathbf{0}$ for any $c \in K$, any finite proof subtree containing an edge labelled with $\mathbf{0}$ has weight $\mathbf{0}$. We illustrate this idea with the following example.

Example 3.3.8. Recall $\mathbb{P}_{gsp}$ from Example 3.3.2, and consider the goal $\mathsf{reachable(c)}$. Its weighted derivation tree $[\![\mathsf{reachable(c)}]\!]_{\mathbb{P}_{gsp}}$ includes a finite path (subtree) witnessing a proof of the goal.

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

Intuitively, this says that c is reachable from the initial state a in in (3.26), with weight 4. However, in $[\![\mathsf{reachable(c)}]\!]_{\mathbb{P}_{gsp}}$ there are also other proof subtrees, which feature cycles, as for instance

$$\mathtt{reach(c)} \xrightarrow{4} \bullet \to \mathtt{reach(a)} \xrightarrow{9} \bullet \to \mathtt{reach(c)} \xrightarrow{4} \bullet \to \mathtt{reach(a)} \xrightarrow{0} \bullet \to \mathtt{init(a)} \xrightarrow{0} \bullet \tag{3.29}$$

$$\mathtt{reach(c)} \xrightarrow{4} \bullet \to \mathtt{reach(a)} \xrightarrow{9} \bullet \to \mathtt{reach(c)} \xrightarrow{4} \bullet \to \mathtt{reach(a)} \xrightarrow{9} \bullet \to \mathtt{reach(c)} \to \cdots \tag{3.30}$$

Our algorithm for computing the weight of $w(\mathsf{reachable(c)})$ will prune both (3.29) and (3.30), letting them both become equal to

$$\mathtt{reach(c)} \xrightarrow{4} \bullet \to \mathtt{reach(a)} \xrightarrow{+\infty} \bullet \tag{3.31}$$

where $+\infty$ is the element $\mathbf{0}$ in the semiring of $\mathbb{P}_{gsp}$. To see that this procedure yields the correct result, note that the weight of the proof subtree (3.31) is $4 + (+\infty) = +\infty$, which does not affect the calculation of the weight $(= 4)$ of $\mathsf{reachable(c)}$.

Now we give the generic algorithms which compute the weight of a goal A given its weighted derivation tree $[\![A]\!]_w$ in the program $\mathbb{W}$. For readability, we divide our approach into 2 algorithms in Figure 3.3.

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

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

Algorithm 3 Prune a weighted derivation tree

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

for path $\pi$ in $\mathcal{T}$ do
    atom_list = [ ]
    for node $v$ in $\pi$ do
        if label($v$) in atom_list then
            $t$ = grandparent of $v$, $u$ = parent of $v$
            cut the descendants of $u$ from $\mathcal{T}$
            re-label the edge $t \to u$ by $\mathbf{0}$
        else
            atom_list.append(label($v$))
return tree $\mathcal{T}$
```

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

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

Prune $\mathcal{T}$ using Algorithm 3
function WEIGHT($x, \mathcal{T}$)
    sum_list = [ ]
    for children $y$ of $x$ do
        prod_list = [label($x \to y$)]                ▷ first get the weight of the clause
        for children $z$ of $y$ do
            prod_list.append(WEIGHT($z, \mathcal{T}$))
            cur_weight = $\prod$ prod_list ▷ calculate the $\cdot$-product of all the values in prod_list
        sum_list.append(cur_weight)
    return $\sum$ sum_list            ▷ return the $+$-sum of all the values in product_list
$s$ = root of $\mathcal{T}$                              ▷ calculate the weight of node $s$ in $\mathcal{T}$
return WEIGHT($s, \mathcal{T}$)
```

Figure 3.3: Algorithms for computing WLP weights

Chapter 4

Functorial semantics for logic programming

In this section we turn to an algebraic perspective of (probabilistic, weighted) logic programs, restricted to the propositional cases. The guiding principle adopted here is the *functorial semantics* approach proposed by Lawvere [Law63], where an algebraic theory is encoded as a syntactic category of terms, and models of this algebraic theory are identified with product-preserving functors from the syntactic category to appropriate semantic categories. In analogy, we make a formal distinction between the syntax and semantics of logic programs, both as categories. This 'separation of concern' has the benefit of clearly isolating the purely inferential structure underlying a program from its 'type' (*i.e.* classical, probabilistic, or weighted, expressed by the semantics). In a nutshell, given a (probabilistic, weighted) logic program $\mathbb{P}$, we freely generate a category Syn of string diagrams as its syntactic category that captures the inferential structure of $\mathbb{P}$. Then the semantics is given by structure-preserving functors from Syn into some other categories that act as the semantic domain of interpretation. The main conceptual contribution is that different flavours of logic programs — classical, probabilistic, weighted — amount to choosing different semantic categories but for the same syntactic category.

The main gain of this perspective is a diagrammatic presentation of some well-known semantic constructs. Thanks to the two-dimensional syntax of string diagrams, one can compose the diagrams representing clauses into more elaborated morphisms, possibly including feedback wires. We provide a diagrammatic description of some common semantics in the logic programming literature, such as immediate consequence operators, least Herbrand models, distribution semantics for probabilistic programs.

Another payoff of this functorial semantics approach is an original perspective on the correspondence between propositional PLP programs and Bayesian networks (Definition 2.1.13). Interestingly, we show that the two-way transformation between (acyclic) PLP programs and Bayesian networks is captured purely at the syntactic level in our framework.

This section is organised as follows. We start with defining the syntactic categories underlying classical, probabilistic and weighted logic programs (Section 4.1). Next we present the functorial semantics of classical logic programs, ad use it to diagrammatically present the Herbrand semantics (Section 4.2). Then we move on to probabilistic logic programs, and study the correspondence with Bayesian networks from a functorial point of view (Section 4.3.1). Finally we discuss the case of weighted logic programming, and present a diagrammatic account of the least-fixed point semantics (Section 4.4.1).

4.1 Syntactic categories

In this subsection we introduce the syntactic categories to be used in all the following discussion on the functorial semantics of logic programs, for classical, probabilistic and weighted case.

The starting point of this syntactic category is the observation that every (classical, probabilistic, weighted) logic program $\mathbb{P}$ has an underlying *definite* logic program *inf*($\mathbb{P}$) (to be formally introduced in Definition 4.1.1) describing the purely inferential structure of $\mathbb{P}$. Intuitively, *inf*($\mathbb{P}$) disregards the information about probabilities, weights, and negated clauses, and only stores which atoms directly determines the values of which atoms. *inf*($\mathbb{P}$) is commonly used in the definition of the *dependency graph* of $\mathbb{P}$ [DSBS02], which describes the dependency relation between atoms in $\mathbb{P}$, and plays a key role in, for example, the definition of stratified logic programs [ABW88]. To incorporate this idea in our approach, we define a 'forgetful' function from arbitrary clauses to definite clauses.

Definition 4.1.1. The forgetful function τ^{gen} (resp. τ^{pr}, τ^{wt}) from arbitrary classical (*resp.* probabilistic, weighted) clauses to definite clauses is defined as follows:

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

The *inference program* of $\mathbb{P}$ is the image of $\mathbb{P}$-clauses under the suitable forgetful function τ, namely *inf*$(\mathbb{P}) = \{\tau(\varphi) \mid \varphi \in \mathbb{P}\}$.

In words, τ drops the quantitative labels and the negation, and only keeps the atom information. We say $\mathbb{P}$ is *based on* a definite logic program $\mathbb{L}$ if *inf*$(\mathbb{P}) = \mathbb{L}$. The resulting syntactic category is closely related with *dependency graphs* of the program.

Definition 4.1.2 ([DSBS02])**.** The *dependency graph* of a (LP, PLP, or WLP) program $\mathbb{P}$ is a directed graph, whose nodes are atoms in $\mathbb{P}$, and there is an edge $B \to A$ if and only if there exists a $\mathbb{P}$-clause whose head is A and body contains B (either positively or negatively).

Note that, if we read the definite clause $A \leftarrow B_1, \ldots, B_k$ as 'nodes $B_1, \ldots, B_k$ are parents of A', then *inf*($\mathbb{P}$) defines exactly (the components of) the dependency graph of $\mathbb{P}$. Observe that there may exist distinct clauses in $\mathbb{P}$ that have the same image under τ, so the size of *inf*($\mathbb{P}$) is less or equal to that of $\mathbb{P}$.

Example 4.1.3. We demonstrate *inf*$(\cdot)$ via the PLP program $\mathbb{P}_{\text{wet}}$ below. It 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)

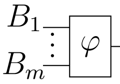
For instance, we can calculate the success probability $\pi_{\mathbb{P}_{\text{wet}}}(\texttt{Winter} \wedge \texttt{WetGrass}) = 0.1434$. Its inference program *inf*$(\mathbb{P}_{\text{wet}})$ consists of the following six clauses:

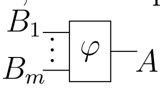
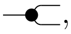
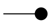
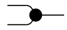
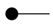
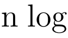
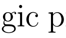
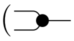
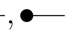
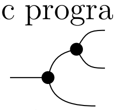
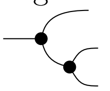
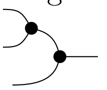
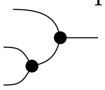
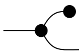
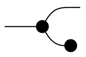
`Winter`	$\leftarrow$.	(φ_1)		`WetGrass`	$\leftarrow$ `Sprinkler.`	(φ_4)
`Sprinkler`	$\leftarrow$ `Winter.`	(φ_2)		`WetGrass`	$\leftarrow$ `Rain.`	(φ_5)
`Rain`	$\leftarrow$ `Winter.`	(φ_3)		`SlipperyRoad`	$\leftarrow$ `Rain.`	(φ_6)

For instance, $\tau^{pr}(\psi_3) = \tau^{pr}(\psi_4) = (\texttt{Rain} \leftarrow \texttt{Winter.}) = \varphi_3$. In particular, note the distinction between $|\mathbb{P}_{\text{wet}}|$ and *inf*$(\mathbb{P}_{\text{wet}})$: the latter is obtained by simply forgetting about all the probabilistic labels, thus containing eight clauses and is not definite.

As hinted by the dependency graphs and their connection with inference programs, such inferential structure of $\mathbb{P}$ can be captured diagrammatically, indeed as a category of string diagrams.

Definition 4.1.4. Given a definite logic program $\mathbb{L}$ over At, We define $\mathsf{Syn}_{\mathbb{L}}$ to be a freely generated CDMU category (see Definition 2.3.10), such that:

- the generating objects are $A \in At$;
- the generating morphisms are string diagrams of the form  one for each clause $B_1, \ldots, B_m \to A$ in $\mathbb{L}$.

In words, objects in $\mathsf{Syn}_{\mathbb{L}}$ are finite lists of atoms, and morphisms are string diagrams freely composed using generating morphisms of the form  (that is sensitive to $\mathbb{L}$) as well as , , ,  (that is irrelevant to $\mathbb{L}$). With some abuse of notation, we use φ to refer to both a clause and its corresponding string diagram in $\mathsf{Syn}_{\mathbb{L}}$ when the context is clear. We choose to work with CDMU categories because their equations subsume structure that is always present in logic programs. Roughly speaking, (, ) reflects the fact that in the logic programming paradigms we consider here one can use as many copies of an atom as one wants; (, ) corresponds to the implicit structure in logic programming to 'bundle together' all the possible derivation of the same atom. 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 equivalent to just working with a single occurrence.

As we mentioned at the beginning, $\mathsf{Syn}_{\mathbb{L}}$ will act as the syntactic category for all (classical, probabilistic, or weighted) $\mathbb{P}$ such that whose inference structure is $\mathit{inf}(\mathbb{P}) = \mathbb{L}$. In the parts on functorial semantics, we will identify logic programs based on $\mathbb{L}$ with structure-preserving functors from $\mathsf{Syn}_{\mathbb{L}}$ to some appropriate 'semantic categories', which will vary depending on whether the program is classical, probabilistic, or weighted.

4.2 Logic programming

In this section we introduce a functorial semantics of classical propositional logic programs, which culminates in Proposition 4.2.5 below. In Subsection 4.2.1on functorial semantics, we provide a semantic category, and identify logic programs with certain *models* — structure-preserving functors from the syntactic category to the semantic category. Then in Subsection 4.2.2 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.

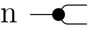
4.2.1 Functorial Semantics of LP

In this subsection we introduce a categorical semantics for classical logic programming, and characterise logic programs as functors from the syntactic categories introduced in Definition 4.1.4 to an appropriate semantic domain. In the current case of classical logic programming, such semantic domain, as a category, should reflect the fact that atoms take Boolean values, and clauses are functions between Boolean states.

Definition 4.2.1. The category $\mathsf{Set}(\mathbb{B})$ has the following components:

- Objects are finite products of two-element Boolean algebras.
- Morphisms are functions between the objects.

In particular, we pick a specific singleton set $\mathbf{1} = \{\bullet\}$ as the 0-ary product object. We write $\vee$, $\wedge$, $(\cdot)^-$ for the standard Boolean algebra operations, which are also morphisms in $\mathsf{Set}(\mathbb{B})$. Note that every $\mathsf{Set}(\mathbb{B})$-object is itself a finite Boolean algebra, with the operations defined pointwise.

Since the syntactic category is a CDMU category, first we need to check that the semantic category has enough structure to interpret the CDMU part of Syn. Note that in the proof of Proposition 4.2.2 we deliberately use the symbolic rather than diagrammatic notation for the CDMU structure in $\mathsf{Set}(\mathbb{B})$ — for instance ∇ rather than  — to distinguish from the syntactic category; but once the functorial semantics is established this distinction is no longer necessary thus we use the more intuitive diagrammatic notation for both.

Proposition 4.2.2. $\mathsf{Set}(\mathbb{B})$ *is a CDMU category.*

Proof. The category $\langle \mathsf{Set}(\mathbb{B}), \times, \mathbf{1}\rangle$ 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 $\mathbb{B}$). We define the CDMU structure on the two-element boolean algebra $\mathbb{B} = \{0, 1\}$ as follows.

$$\begin{array}{rrcl}\nabla_{\mathbb{B}}\colon & \mathbb{B} & \to & \mathbb{B}\times\mathbb{B} \\ & x & \mapsto & (x,x)\end{array} \qquad \begin{array}{rrcl}\epsilon_{\mathbb{B}}\colon & \mathbb{B} & \to & \mathbf{1} \\ & x & \mapsto & *\end{array} \qquad \begin{array}{rrcl}\Delta_{\mathbb{B}}\colon & \mathbb{B}\times\mathbb{B} & \to & \mathbb{B} \\ & (x,y) & \mapsto & x\vee y\end{array} \qquad \begin{array}{rrcl}\eta_{\mathbb{B}}\colon & \mathbf{1} & \to & \mathbb{B} \\ & * & \mapsto & 0\end{array}$$

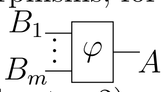
We only verify the equations for the monoid structure. Given arbitrary $x, y, z \in \mathbb{B}$ (below we omit the obvious subscripts for readability),

- $((id \otimes \eta)\ ;\ \Delta)(x) = \Delta(x, 0) = x \vee 0 = x$. Similarly $(\Delta \otimes id)\ ;\ \Delta = id$.
- $((\Delta \otimes id)\ ;\ \Delta)(x, y, z) = \Delta(x \vee y, z) = x \vee y \vee z = ((id \otimes \Delta)\ ;\ \Delta)(x, y, z)$.
- $(\gamma\ ;\ \Delta)(x, y) = \Delta(y, x) = x \vee y = \Delta(x, y)$.

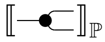
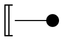
The CDMU structure on arbitrary objects of the form $\mathbb{B}_1 \times \cdots \times \mathbb{B}_k$ are defined pointwise, and it follows immediately that they satisfy the CDMU equations. □

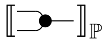
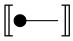
Now we are ready to establish the functorial semantics. Let us fix a classical logic program $\mathbb{P}$ over At, with underlying inference structure being $\mathbb{L} := \mathit{inf}(\mathbb{P})$. The goal is to associate $\mathbb{P}$ with a functor of the form $\mathsf{Syn}_{\mathbb{L}} \to \mathsf{Set}(\mathbb{B})$ that fully captures the information of $\mathbb{P}$.

Definition 4.2.3. The CDMU functor $[\![-]\!]_{\mathbb{P}} : \mathsf{Syn}_{\mathbb{L}} \to \mathsf{Set}(\mathbb{B})$ is defined as follows using the free construction of the domain category $\mathsf{Syn}_{\mathbb{L}}$:

- On objects, $[\![-]\!]_{\mathbb{P}}$ maps each $A \in At$ to $\mathbb{B}_A = \{0_A, 1_A\}$.
- On morphisms, for each $\mathbb{L}$-clause $\varphi \equiv B_1, \ldots, B_k \to A$ and its associated generating morphism , $[\![\varphi]\!]_{\mathbb{P}}$ maps a state $u \in \mathbb{B}_{B_1} \times \cdots \times \mathbb{B}_{B_k}$ to 1_A if u as an interpretation (*cf.* Chapter 2) satisfies $u \vDash \mathsf{body}(\psi)$ for some $\psi \in \mathbb{P}$ where $\tau^{gen}(\psi) = \varphi$, and to 0_A otherwise.

It is helpful to explicitly write down the behaviour of the functor $[\![-]\!]_{\mathbb{P}}$ on the CDMU structure of $\mathsf{Syn}_{\mathbb{L}}$. In a nutshell, it maps the CDMU structure to the CDMU structure, where the latter guaranteed by Proposition 4.2.2. More precisely,

$[\![$  $]\!]_{\mathbb{P}}\ :\ \mathbb{B} \to \mathbb{B}\times\mathbb{B}$, $\ x \mapsto (x, x)$ $\qquad [\![$  $]\!]_{\mathbb{P}}\ :\ \mathbb{B} \to \mathbf{1}$, $\ x \mapsto \bullet$

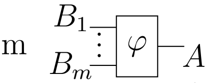
$[\![$  $]\!]_{\mathbb{P}}\ :\ \mathbb{B}\times\mathbb{B} \to \mathbb{B}$, $\ (x_1, x_2) \mapsto x_1 \wedge x_2$ $\qquad [\![$  $]\!]_{\mathbb{P}}\ :\ \mathbf{1} \to \mathbb{B}$, $\ \bullet \mapsto 1$

Example 4.2.4. Let us look at the clause $\psi \equiv A \leftarrow B_1, \neg B_2$. Its inference structure is captured by the definite clause $\varphi \equiv A \leftarrow B_1, B_2$. Moreover, ψ is interpreted as a function $[\![\varphi]\!] : \mathbb{B}_{B_1} \times \mathbb{B}_{B_2} \to \mathbb{B}_A$ mapping, for instance, $(1_{B_1}, 0_{B_2})$ to 1_A and $(1_{B_1}, 1_{B_2})$ to 0_A. Incidentally, note that $[\![\varphi]\!]$ is *not* a Boolean function (*i.e.* it does not respect the lattice structure), as for instance $(1_{B_1}, 0_{B_2}) \vee (1_{B_1}, 0_{B_2}) = (1_{B_1}, 1_{B_2})$ but $[\![\varphi]\!]\,(1_{B_1}, 1_{B_2}) \vee [\![\varphi]\!]\,(1_{B_1}, 1_{B_2}) \neq [\![\varphi]\!]\,(1_{B_1}, 1_{B_2})$. This explains why we cannot choose to interpret in the category of finite Boolean algebras and Boolean functions between them.

The next proposition formally states the correspondence characterising the functorial semantics of logic programs. In there, we say a functor $\mathcal{F}\colon \mathbb{C} \to \mathbb{D}$ *preserves generating objects* if it maps generating objects of $\mathbb{C}$ to generating objects of $\mathbb{D}$ (assuming the objects of each of these two categories are freely obtained from some sets of generators, respectively). In our case, our choice of $\mathbb{C}$ and $\mathbb{D}$ are $\mathsf{Syn}_{\mathbb{L}}$ and $\mathsf{Set}(\mathbb{B})$, respectively, by noticing that $\mathsf{Set}(\mathbb{B})$ has $\mathbb{B}$ as the generating objects. Then the requirement of preserving generating objects ensures that a functor maps each atom $A \in At$ — seen as object of $\mathsf{Syn}_{\mathbb{L}}$ — to a two-element Boolean algebra $\mathbb{B}$, which we write $\mathbb{B}_A = \{0_A, 1_A\}$ to emphasise this correspondence.

Proposition 4.2.5. *There is a* 1-1 *correspondence between logic programs based on* $\mathbb{L}$ *and CDMU functors of type* $\mathsf{Syn}_{\mathbb{L}} \to \mathsf{Set}(\mathbb{B})$ *that preserve generating objects.*

Proof. The construction of a generator-preserving CDMU functor $[\![-]\!]_{\mathbb{P}}$ from a program $\mathbb{P}$ is already discussed in Definition 4.2.3.

For the other direction, given a CDMU functor $\mathcal{G}\colon \mathsf{Syn}_{\mathbb{L}} \to \mathsf{Set}(\mathbb{B})$ that preserves generating objects, we construct a logic program $Prog(\mathcal{G})$. We know that for each $A \in At$, $\mathcal{G}(A)$ is some copy of $\mathbb{B}$, which we denote as $\mathbb{B}_A$. For each generating morphism  of $\mathsf{Syn}_{\mathbb{L}}$, $\mathcal{G}(\varphi)$ is a function of the form $\mathbb{B}_{B_1} \times \cdots \times \mathbb{B}_{B_m} \to \mathbb{B}_A$. We assume that the support of $\mathcal{G}(\varphi)$ — the set of all v such that $\mathcal{G}(\varphi)(v) = 1_A$ — is $\{v_1, \ldots, v_d\}$. Then we construct d-many clauses $\varphi_1, \ldots, \varphi_d$ such that $\tau^{gen}(\varphi_j) = A \leftarrow B_1, \ldots, B_m$, one for each v_i. In particular, suppose v_i satisfies that $v_i(p_1) = 1_{B_{p_1}}, \ldots, v_i(p_k) = 1_{B_{p_k}}$, $v_i(q_1) = 0_{B_{q_1}}, \ldots, v_i(q_\ell) = 0_{B_{q_\ell}}$ (with $k + \ell = m$ and $\{p_1, \ldots, p_k, q_1, \ldots, q_\ell\} = \{1, \ldots, m\}$), then φ_i is defined as $A \leftarrow B_{p_1}, \ldots, B_{p_k}, \neg B_{q_1}, \ldots, \neg B_{q_\ell}$. The program $Prog(\mathcal{G})$ is then the collection of all the clauses constructed from each generating morphism for $\mathsf{Syn}_{\mathbb{L}}$.

Next we show that the aforementioned two constructions are inverse to each other, thus witnessing the isomorphism between logic programs and functors as in the statement.

- We start from a logic program $\mathbb{P}$ based on $\mathsf{Syn}_{\mathbb{L}}$. For an arbitrary clause $\varphi \equiv A \leftarrow B_1, \ldots, B_m.$ in $\mathbb{L}$, suppose $\{\varphi_1, \ldots, \varphi_d\}$ are all the clauses φ_j such that $\tau^{gen}(\varphi_j) = \varphi$, then $[\![\varphi]\!]_{\mathbb{P}}$ is defined as a function of type $\mathbb{B}_{B_1} \times \cdots \times \mathbb{B}_{B_m} \to \mathbb{B}_A$ whose support (namely the set $\mathcal{G}(\varphi)^{-1}(1_A)$) has size d: each element in the support corresponds to the body of one of those d-many clauses. Then from $[\![\varphi]\!]_{\mathbb{P}}$ we retrieve d clauses, which are exactly $\varphi_1, \ldots, \varphi_d$: if $v \in \mathbb{B}_{B_1} \times \cdots \times \mathbb{B}_{B_m}$ satisfies $[\![\varphi]\!]_{\mathbb{P}}\,(v) = 1_A$, then there exists some φ_j such that for all $i = 1, \dots, m$, $v(B_i) = 1_{B_i}$ 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 $\mathcal{G}\colon \mathsf{Syn}_{\mathbb{L}} \to \mathsf{Set}(\mathbb{B})$. Fix some $\mathbb{L}$-clause φ, and suppose $\mathcal{G}(\varphi)^{-1}(1_A) = \{v_1, \dots, v_d\}$. This means $\mathcal{G}$ determines d-many clauses $\varphi_1, \dots, \varphi_d$ whose images under τ^{gen} are all φ. Then the action of the functor $[\![-]\!]_{Prog(\mathcal{G})} : \mathsf{Syn}_{\mathbb{L}} \to \mathsf{Set}(\mathbb{B})$ on $\varphi \in \mathbb{L}$ is totally determined by $\{\varphi_1, \dots, \varphi_d\}$: $[\![\varphi]\!]_{Prog(\mathcal{G})}(v) = 1_A$ 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 $[\![\varphi]\!]_{Prog(\mathcal{G})}(v) = 1_A$ if and only if $v = v_j$, for some $v_j \in \{v_1, \dots, v_d\}$. This means that $[\![\varphi]\!]_{Prog(\mathcal{G})} = \mathcal{G}(\varphi)$, for arbitrary $\mathbb{L}$-clause φ.

Therefore we can conclude the bijection between logic programs based on $\mathbb{L}$ and generator-preserving CDMU functors $\mathsf{Syn}_{\mathbb{L}} \to \mathsf{Set}(\mathbb{B})$. □

Echoing the terminology of Lawvere's functorial semantics [Law63], we will refer to the functors as in Proposition 4.2.5 simply as *models* of the syntactic theory $\mathsf{Syn}_{\mathbb{L}}$. When we want to emphasise that such models are that corresponding to classical logic programs, we will say it is a model in $\mathsf{Set}(\mathbb{B})$.

4.2.2 Diagrammatic representations of some semantic constructs

One useful outcome of the functorial semantics in general is the possibility of capturing some algebraic properties solely in the syntactic categories. In our case, the functorial semantics established so far enables us to present some well-known semantic concepts of logic programs using the diagrammatic language. More precisely, we will study string diagrams in the syntactic category whose images under the model are precisely the semantics that we are interested in. In this subsection we restrict ourselves to two simple yet fundamental semantic concepts of classical logic programming: the immediate consequence operator, and the consequence operator for definite programs (with the least Herbrand semantics as a special case), which we refer to Section 2 for an overview.

Immediate consequence operator

We begin with the immediate consequence operator $\mathbf{T}$ (*cf.* Definition 2.1.7), a basic yet fundamental concept beneath many denotational semantics of logic programs [Gel87, ABW88, GL88, VGRS91a]. Let us fix a logic program $\mathbb{P}$ over atom set $At = \{A_1, \dots, A_n\}$, and denote its inference structure as $\mathbb{L}$, namely $\mathbb{L} = \mathit{inf}(\mathbb{P})$. We provide an example below, both to recall immediate consequence operators and to serve as a running-example of this part.

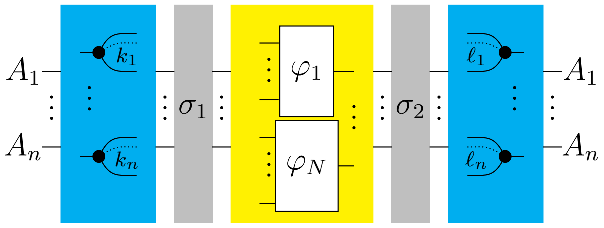
Example 4.2.6. We consider an example from [GL88]. The atom set At_{pq} consists of $p(1,1)$, $p(1,2)$, $p(2,1)$, $p(2,2)$, $q(1)$, $q(2)$, and the program $\mathbb{P}_{pq}$ contains the following six clauses:

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

Then the immediate consequence operator $\mathbf{T}_{\mathbb{P}_{pq}}$ is a function of the form $\mathbb{B}_{At_{pq}} \to \mathbb{B}_{At_{pq}}$, or equivalently $\mathcal{P}(At_{pq}) \to \mathcal{P}(At_{pq})$. We use the latter type to simplify the notation used in the remainder of this example. For instance, maps both $\varnothing$ and $\{p(1,2), q(2)\}$ to $\{p(1,2), p(2,1)\}$ (via clauses ψ_1 and ψ_2), and maps $\{p(1,2)\}$ to $\{p(1,2), p(2,1), q(1)\}$ (via clauses ψ_1, ψ_2 and ψ_4).

As stated in the very beginning of this subsection, the goal is to express $\mathbf{T}_{\mathbb{P}}$ via our diagrammatic language as certain string diagram, in the sense that its semantics under the model $[\![-]\!]_{\mathbb{P}}$ is precisely $\mathbf{T}_{\mathbb{P}}$. Intuitively, to capture that the immediate consequence operator uses every single clause exactly once in parallel, we put side-by-side all the generating morphisms for $\mathsf{Syn}_{\mathbb{L}}$, as each one of such generating morphism represent one clause in $\mathbb{L}$.

Definition 4.2.7. The *immediate consequence diagram* $T_{\mathbb{L}}$ for $\mathbb{P}$ is defined as the following morphism of type $[A_1, \ldots, A_n] \to [A_1, \ldots, A_n]$ in $\mathsf{Syn}_{\mathbb{L}}$:



(4.1)

The left and the right ports both consist of all the atoms $A_1, \ldots, A_n$ in At. For each A_i, suppose there are k_i-many $\mathbb{L}$-clauses whose bodies include A_i and ℓ_i-many $\mathbb{L}$-clauses whose heads are A_i, then we construct k_i copies (via ─●⊏ in the left blue box) and ℓ_i cocopies (via ⊐●─ in the right blue box) of A_i. The yellow box contains the parallel composition of the associated string diagrams of all the $\mathbb{L}$-clauses $\varphi_1, \ldots, \varphi_N$. In the two grey boxes, there are suitably many swapping morphisms ⊐⊏ to match each copy/cocopy of A_i with an input/output wire A_i in the yellow box.

The immediate consequence diagram expresses the immediate consequence operator diagrammatically in the following sense.

Proposition 4.2.8. *The immediate consequence diagram expresses the immediate consequence operator under interpretation* $[\![-]\!]_{\mathbb{P}}$*:*

$$[\![T_{\mathbb{L}}]\!]_{\mathbb{P}} = \mathbf{T}_{\mathbb{P}}$$

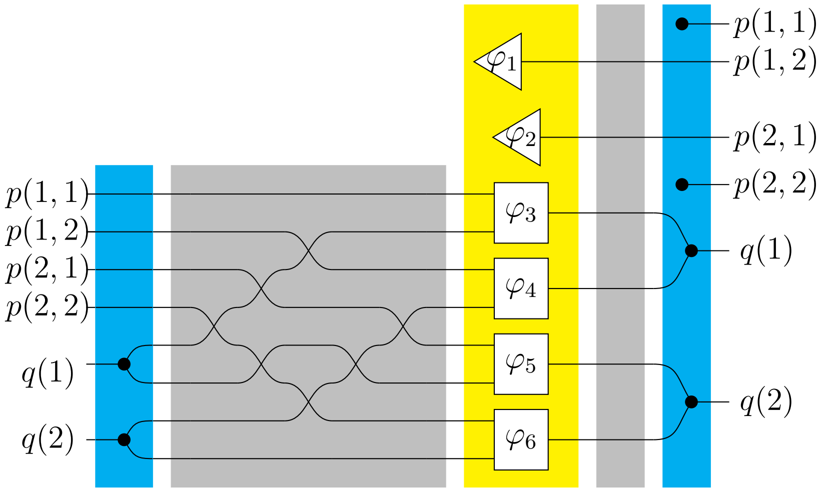
Proof. It suffices to show that for arbitrary $u \in \mathbb{B}_{A_1} \times \cdots \times \mathbb{B}_{A_n}$ and $A \in At$, $[\![T_{\mathbb{L}}]\!]_{\mathbb{P}}(u)(A) = 1_A$ if and only if $\mathbf{T}_{\mathbb{P}}(u)(A) = 1_A$.

By definition $\mathbf{T}_{\mathbb{P}}(u)(A) = 1_A$ 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 $[\![\varphi]\!]_{\mathbb{P}} : u|_{\varphi} \mapsto 1_A$, where $u|_{\varphi}$ is the projection of u (as a tuple of values) to those atoms in $\mathsf{body}(\varphi)$. This means that $\mathbf{T}_{\mathbb{P}}(u)(A) = 1_A$ if and only if there exists $\varphi \in \mathit{inf}(\mathbb{P})$ such that $\mathsf{head}(\varphi) = A$ and $[\![\varphi]\!]_{\mathbb{P}} : u|_{\varphi} \mapsto 1_A$. By the interpretation of —●⊏ and ⊐●— under $[\![-]\!]_{\mathbb{P}}$, this again is equivalent to that $[\![T_{\mathbb{L}}]\!]_{\mathbb{P}}(u)(A) = 1_A$. □

Example 4.2.9. The inference structure $\mathit{inf}(\mathbb{P}_{pq})$ of $\mathbb{P}_{pq}$ in Example 4.2.6 consists of the following 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 of $\mathsf{Syn}_{\mathbb{L}_{pq}}$. Then the immediate consequence diagram $T_{\mathbb{L}_{pq}}$ for $\mathbf{T}_{\mathbb{P}_{pq}}$ is depicted as follows, where we colour the diagrams according to that in Definition 4.2.7:

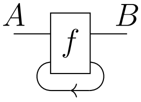


Interlude: traced extension

Before considering the next semantic constructs of consequence operators and Herbrand models, we point out that these are *fixed point-style* semantics for *definite* logic programs. Therefore to provide the same kind of analysis as for the immediate consequence operator, we need to mildly extend our string diagrammatic language $\mathsf{Syn}_{\mathbb{L}}$ to include 'feedback wires' (formally called 'traces', see Definition 2.3.7), and also extend the semantic category accordingly. While we continue with the setting of program $\mathbb{P}$ of this section, we add the further assumption that $\mathbb{P}$ is definite, namely has not negation in the body. Although in this case $\mathbb{L} := \mathit{inf}(\mathbb{P}) = \mathbb{P}$, we still keep to use $\mathbb{L}$ and $\mathbb{P}$ separately to highlight the difference between syntactic and semantic components. The necessity of such distinction is clearer in the latter discussion on probabilistic and weighted programs.

Definition 4.2.10. The syntactic category $\mathsf{Syn}^{tr}_{\mathbb{L}}$ is the free traced CMUD category $\mathsf{freeTrCDMU}(At, \Sigma_{\mathbb{L}})$, namely:

- The objects are finite lists over At.

- The morphisms are string diagrams from $\mathsf{freeCDMU}(At, \Sigma_{\mathbb{L}})$ plus the possibility of adding traces of the form  , modulo the axioms for traced categories (see Definition 2.3.7).

Note that we purposefully choose not to include traces in the 'basic' syntactic category $\mathsf{Syn}_{\mathbb{L}}$ (Sec. 4.1), out of three reasons. First, we consider fixed points as a feature specific to certain semantic constructs, rather than a primitive construction of logic program syntax itself. Second, for a traced syntactic category one also needs a traced semantic category to properly interpret the syntactic objects, which is not met by $\mathsf{Set}(\mathbb{B})$ essentially because clauses involving negations are not monotone in general, nor met by the semantic category we choose later for interpreting PLP. Third, we point out that traces are not required for developments in case one restricts attention to *acyclic* programs. This is mostly evident in the probabilistic case (see Subsection 4.3.3), where we relate PLP programs with Bayesian networks.

Turning to the semantics, since the category $\mathsf{Set}(\mathbb{B})$ used to interpret $\mathsf{Syn}_{\mathbb{L}}$-morphisms is not traced (similar to the fact that Set is not), we need to turn to another semantic category that is both traced and close enough to $\mathsf{Set}(\mathbb{B})$ to retain the functorial semantics established in Subsection 4.2.1. We choose the subcategory $\mathsf{MonFunc}(\mathbb{B})$ of $\mathsf{Set}(\mathbb{B})$ whose morphisms are restricted to *monotone* functions regarding the ordering $1 \geq 0$. Thanks to the fact that definite clauses have no negation in their bodies, they can be read as monotone functions of the form $\prod_{B \in \mathsf{body}(\varphi)} \mathbb{B}_B \to \mathbb{B}_{\mathsf{head}(\varphi)}$ that maps only $(1, \ldots, 1)$ to 1. So they are morphisms in this category of monotone functions.

Definition 4.2.11. $\mathsf{MonFunc}(\mathbb{B})$ is the subcategory of $\mathsf{Set}(\mathbb{B})$ where morphisms are restricted to monotone functions $\mathbb{B}^m \to \mathbb{B}^n$.

This new semantic category $\mathsf{MonFunc}(\mathbb{B})$ is also a CDMU category because the morphisms forming the SMC and CDMU structure of $\mathsf{Set}(\mathbb{B})$ are all monotone. Moreover, it has enough structure to interpret feedback in the novel syntactic category $\mathsf{Syn}^{tr}_{\mathbb{L}}$.

Proposition 4.2.12. *The category* $\mathsf{MonFunc}(\mathbb{B})$ *is traced.*

Proof. We give a direct proof which verifies every axioms of traced monoidal categories. A simpler proof is provided later using the compact structure of $\mathsf{MonRel}_{\mathbb{B}}$.

We first define the family of functions Tr in $\mathsf{Set}(\mathbb{B})$. Given $f \colon X \times U \to Y \times U$ in $\mathsf{MonFunc}(\mathbb{B})$, for arbitrary $x \in X$, let u_x be the smallest $u \in U$ such that $\pi_2 \circ f(x, u) = u$, whose existence is guaranteed by the fact that $\pi_2 \circ f(x, \cdot)$ is a monotone function $U \to U$. Then define

$$Tr^U_{X,Y}(x) = (\pi_1 \circ f)(x, u_x) \tag{4.2}$$

Now we verify all the traced monoidal category conditions for such Tr. Fix some $f \colon X \times U \to Y \times U$.

1. Naturality in X. Given an arbitrary $g\colon X' \to X$, we show that $Tr^U_{X,Y}(f) \circ g = Tr^U_{X',Y}(f \circ (g \times id_U))$, both as functions $X' \to Y$. Pick an arbitrary $x' \in X'$. Let $u_1 = \mu u.\pi_2 \circ f(g(x'), u)$. Then $Tr^U_{X,Y}(f) \circ g(x') = \pi_1 \circ f(g(x'), u_1)$. Let $u_2 = \mu u.\pi_2 \circ f \circ (g \times id_U)(x', u)$, then $Tr^U_{X',Y}(f \circ (g \times id_U))(x') = \pi_1 \circ f \circ (g \times id_U)(x', u_2)$.

   It suffices to show that $u_1 = u_2$, because in that case, $\pi_1 \circ f(g(x'), u_1) = \pi_1 \circ f(g(x'), u_2) = \pi_1 \circ f \circ (g \times id_U)(x', u_2)$. And to show $u_1 = u_2$, we observe that they satisfies the fixed-point equation of each other. On one hand, $\pi_2 \circ f \circ (g \times id_U)(x', u_1) = \pi_2 \circ f(g(x'), u_1) = u_1$. On the other hand, $\pi_2 \circ f(g(x'), u_2) = \pi_2 \circ f \circ (g \times id_U)(x', u_2) = u_2$.

2. Natuality in Y. Given an arbitrary $h\colon Y \to Y'$, we show that $h \circ Tr^U_{X,Y}(f) = Tr^u_{X,Y'}((h \times id_U) \circ f)$, both as functions $X \to Y'$. We start with an arbitrary $x \in X$. Let $u_1 = \mu u.\pi_2 \circ f(x, u)$, then $h \circ Tr^U_{X,Y}(f)(x) = h \circ \pi_1 \circ f(x, u_1)$. Let $u_2 = \mu u.\pi_2 \circ (h \times id_U) \circ f(x, u)$. Then $Tr^U_{X,Y'}((h \times id_U) \circ f) = \pi_1 \circ (h \times id_U) \circ f(x, u_2)$. It suffices to show that $u_1 = u_2$, since then $h \circ \pi_1 \circ f(x, u_1) = h \circ \pi_1 \circ f(x, u_2) = \pi_1 \circ (h \times id_U) \circ f(x, u_2)$. For this, we again show that u_1 and u_2 are fixed points of the operator for each other. This follows by observing that $u_2 = \mu u.\pi_2 \circ (h \times id_U) \circ f(x, u) = \mu u.\pi_2 \circ f(x, u) = u_1$.

3. Dinaturality in U. Suppose f is of type $X \times U \to Y \times U'$, and $d\colon U' \to U$. We show that $Tr^{U'}_{X,Y}(f \circ (id_X \times d)) = Tr^U_{X,Y}((id_Y \times d) \circ f)$. The LHS is

$$Tr^{U'}_{X,Y}(f \circ (id_X \times d))(x) = \pi_1 \circ (f \circ (id_X \times d))(x, u'_0) = \pi_1 \circ f(x, d(u'_0))$$

   where $u'_0 = \mu u'.\pi_2 \circ f \circ (id_X \times d)(x, u')$. The RHS is

$$Tr^U_{X,Y}((id_Y \times d) \circ f)(x) = \pi_1 \circ (id_Y \times d) \circ f(x, u_0) = \pi_1 \circ f(x, u_0)$$

   where $u_0 = \mu u.\pi_2 \circ (id_Y \times d) \circ f(x, u)$. It suffices to show that $u_0 = d(u'_0)$, as this entails that

$$\text{LHS} = \pi_1 \circ f(x, d(u'_0)) = \pi_1 \circ f(x, u_0) = \text{RHS}$$

   On one hand, $d(u'_0) \geq u_0$ because $d(u'_0)$ satisfies the defining fixed-point equation of u_0:

$$\begin{aligned} \pi_2 \circ (id_Y \times d) \circ f(x, d(u'_0)) &= d \circ \pi_2 \circ f(x, d(u'_0)) \\ &= d \circ \pi_2 \circ f \circ (id_X \times d)(x, u'_0) \\ &= d(u'_0) \end{aligned}$$

   On the other hand, to prove that $d(u'_0) \leq u_0$, we note that

$$\begin{aligned} d(u'_0) &= d(\bigwedge\{u' \in U' \mid \pi_2 \circ f(x, d(u')) \leq u'\}) \\ &\leq \bigwedge\{d(u') \mid u' \in U' \;\&\; \pi_2 \circ f(x, d(u')) \leq u'\} \qquad \text{(Proposition 4.2.13)} \end{aligned}$$

$$u_0 = \bigwedge\{u \in U \mid \pi_2 \circ (id_Y \times d) \circ f(x,u) \leq u\}$$

Let $\Psi = \{u' \in U' \mid \pi_2 \circ f(x, d(u')) \leq u'\}$ and $\Phi = \{u \in U \mid \pi_2 \circ (id_Y \times d) \circ f(x,u) \leq u\}$, it suffices to show that for every $u \in \Phi$, there exists $u' \in U'$ such that $d(u') \leq u$. Indeed, we can let $u' = \pi_2 \circ f(x,u)$.

$$\begin{aligned} d(u') &= d(\pi_2 \circ f(x,u)) \\ &= \pi_2 \circ (id_Y \times d) \circ f(x,u) \\ &\leq u \end{aligned}$$

And $u' \in \Psi$ because

$$\begin{aligned} \pi_2 \circ f(x, d(u')) &= \pi_2 \circ f \circ (id_X \times d)(x, u') \\ &= \pi_2 \circ f \circ (id_X \times d)(x, \pi_2 \circ f(x,u)) \\ &= \pi_2 \circ f(x, d \circ \pi_2 \circ f(x,u)) \\ &= \pi_2 \circ f(x, \pi_2 \circ (id_Y \times d) \circ f(x,u)) \\ &\leq \pi_2 \circ f(x,u) \\ &= u' \end{aligned}$$

4. Vanishing (i). Suppose $U = I = \mathbf{1}$, namely $f \colon X \times \mathbf{1} \to Y \times \mathbf{1}$, we show that $Tr^{\mathbf{1}}_{X,Y}(f) = \rho_Y \circ f \circ \lambda_X$. Given an arbitrary $x \in X$, $\mu u.\pi_2 \circ f(x,u) = \bullet$, the only element in $\mathbf{1}$. Thus $Tr^{\mathbf{1}}_{X,Y}(f)(x) = \pi_1 \circ f(x, \bullet) = (\rho_Y \circ f \circ \lambda_X)(x, \bullet)$.

5. Vanishing (ii). Assume $f \colon X \times U \times V \to Y \times U \times V$, we show that $Tr^{U}_{X,Y}(Tr^{V}_{X\times U, Y\times U}(f)) = Tr^{U\times V}_{X,Y}(f)$. Fix an arbitrary x.

$$\begin{aligned} & Tr^{U}_{X,Y}(Tr^{V}_{X\times U,Y\times U}(f))(x) \\ =& \pi_1 \circ Tr^{V}_{X\times U,Y\times U}(f)(x, \mu u.\pi_2 \circ Tr^{V}_{X\times U,Y\times U}(f)(x,u)) \\ =& \pi_1 \circ Tr^{V}_{X\times U,Y\times U}(f)(x, \mu u.\pi_2 \circ \pi_{1,2} \circ f(x,u,\mu v.\pi_3 \circ f(x,u,v))) \end{aligned}$$

Denote $\mu u.\pi_2 \circ \pi_{1,2} \circ f(x,u,\mu v.\pi_3 \circ f(x,u,v))$ as u_0, and we have

$$u_0 = \pi_1 \mu(u,v).\pi_{2,3} f(x,u,v)$$

Then continuing the previous calculation we have:

$$\begin{aligned} & \pi_1 \circ Tr^{V}_{X\times U,Y\times U}(f)(x, \mu u.\pi_2 \circ \pi_{1,2} \circ f(x,u,\mu v.\pi_3 \circ f(x,u,v))) \\ =& \pi_1 \circ Tr^{V}_{X\times U,Y\times U}(f)(x, u_0) \\ =& \pi_1 \circ \pi_{1,2} f(x, u_0, \mu v.f(x,u_0,v)) \end{aligned}$$

$$
\begin{aligned}
&=\pi_1 \circ f(x, \pi_1\mu(u,v).\pi_{2,3}f(x,u,v), \pi_2\mu(u,v).\pi_{2,3}f(x,u,v))\\
&=\pi_1 \circ f(x, \mu(u,v).\pi_{2,3}f(x,u,v))\\
&=\mathit{Tr}^{U\times V}_{X,Y}(f)(x)
\end{aligned}
$$

6. Superposing. Given some $e\colon W \to Z$, we verify that $e \times \mathit{Tr}^U_{X,Y}(f) = \mathit{Tr}^U_{W\times X, Z\times Y}(e \times f)$, both functions of type $W \times X \to Z \times Y$. So we pick an arbitrary $(w,x) \in W \times X$.

$$
\begin{aligned}
&\mathit{Tr}^U_{W\times X,Z\times Y}(e\times f)(w,x)\\
=&\pi_{1,2}(e\times f)(w,x,\mu u.\pi_3(e\times f)(w,x,u))\\
=&\pi_{1,2}(e\times f)(w,x,\mu u.\pi_3(e(w),f(x,u)))\\
=&\pi_{1,2}(e\times f)(w,x,\mu u.\pi_2\circ f(x,u))\\
=&\pi_{1,2}(e(w),f(x,\mu u.\pi_2\circ f(x,u)))\\
=&(e(w),\pi_1 f(x,\mu u.\pi_2\circ f(x,u)))\\
=&(e(w),\mathit{Tr}^U_{X,Y}(f)(x))\\
=&(e\times \mathit{Tr}^U_{X,Y}(f))(w,x)
\end{aligned}
$$

7. Yanking. We verify that $\mathit{Tr}^X_{X,X}(\supset\!\!\subset_X) = id_X$. Fix some $x \in X$. Note that $\mu z.\pi_2 \circ \supset\!\!\subset(x,z) = x$, so $\mathit{Tr}^X_{X,X}(\supset\!\!\subset_X) = \pi_1 \circ \supset\!\!\subset_X(x, \mu z.\pi_2 \circ \supset\!\!\subset(x,z)) = \pi_1(x,x) = x$.

This completes the verification of all the axioms for $\mathsf{MonFunc}(\mathbb{B})$ being traced. □

Proposition 4.2.13. *Every* $\mathsf{MonFunc}(\mathbb{B})$ *morphism* $f\colon \mathbb{B}^m \to \mathbb{B}^n$ *preserves* $\wedge$ *in one direction:* $f(\mathbf{a}\wedge\mathbf{b}) \leq f(\mathbf{a}) \wedge f(\mathbf{b})$.

Proof. We know that every monotone function between finite products of $\mathbb{B}$ is defined using $\wedge$ and $\vee$, namely all monotone functions $\mathbb{B}^m \to \mathbb{B}$ can inductively defined as follows:

$$
f ::= \mathbf{0} \mid \mathbf{1} \mid \pi_i,\ \text{where } i \in \{1,\dots,m\} \mid f \wedge f \mid f \vee f \tag{4.3}
$$

thus we define by induction on the structure of f.

The base case of the constant functions $\mathbf{0}$ and $\mathbf{1}$ are trivial, so are the projection cases. We focus on the induction step.

If $f = f_1 \wedge f_2$, then for arbitrary $\mathbf{a}, \mathbf{b} \in \mathbb{B}^m$,

$$
\begin{aligned}
f(\mathbf{a}\wedge\mathbf{b}) &= f_1(\mathbf{a}\wedge\mathbf{b}) \wedge f_2(\mathbf{a}\wedge\mathbf{b})\\
&\leq (f_1(\mathbf{a})\wedge f_1(\mathbf{b})) \wedge (f_2(\mathbf{a})\wedge f_2(\mathbf{b})) && \text{(IH)}\\
&= (f_1(\mathbf{a})\wedge f_2(\mathbf{a})) \wedge (f_1(\mathbf{b})\wedge f_2(\mathbf{b}))\\
&= (f_1\wedge f_2)(\mathbf{a}) \wedge (f_1\wedge f_2)(\mathbf{b})\\
&= f(\mathbf{a}) \wedge f(\mathbf{b})
\end{aligned}
$$

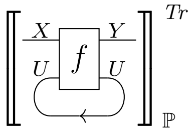
If $f = f_1 \vee f_2$, then for arbitrary $\mathbf{a}, \mathbf{b} \in \mathbb{B}^m$,

$$\begin{aligned}
f(\mathbf{a} \wedge \mathbf{b}) &= (f_1 \vee f_2)(\mathbf{a} \wedge \mathbf{b}) \\
&= f_1(\mathbf{a} \wedge \mathbf{b}) \vee f_2(\mathbf{a} \wedge \mathbf{b}) \\
&\leq (f_1(\mathbf{a}) \wedge f_1(\mathbf{b})) \vee (f_2(\mathbf{a}) \wedge f_2(\mathbf{b})) && \text{(IH)} \\
&= (f_1(\mathbf{a}) \vee f_2(\mathbf{a})) \wedge (f_1(\mathbf{a}) \vee f_2(\mathbf{b})) \wedge (f_1(\mathbf{b}) \vee f_2(\mathbf{a})) \wedge (f_1(\mathbf{b}) \vee f_2(\mathbf{b})) \\
&\leq (f_1(\mathbf{a}) \vee f_2(\mathbf{a})) \wedge (f_1(\mathbf{b}) \vee f_2(\mathbf{b})) \\
&= (f_1 \vee f_2)(\mathbf{a}) \wedge (f_1 \vee f_2)(\mathbf{b}) \\
&= f(\mathbf{a}) \wedge f(\mathbf{b})
\end{aligned}$$

Therefore we can conclude that $f(\mathbf{a} \wedge \mathbf{b}) \leq f(\mathbf{a}) \wedge f(\mathbf{b})$. □

Again, we can establish a functorial semantics for definite logic program $\mathbb{P}$ for the semantic category. Since $\mathsf{Syn}^{tr}_{\mathbb{L}}$ just adds $\mathsf{Syn}_{\mathbb{L}}$ with traces, the construction of this functor is to simply equip $[\![-]\!]_{\mathbb{P}}$ with canonical interpretation of traces.

Definition 4.2.14. A definite logic program $\mathbb{P}$ uniquely determines a traced CDMU functor $[\![-]\!]^{Tr}_{\mathbb{P}}$:

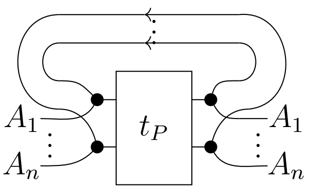
- for every generating object and morphism, $[\![-]\!]^{Tr}_{\mathbb{P}}$ behaves the same as $[\![-]\!]_{\mathbb{P}}$;
- for traced diagrams,  $= Tr^{U}_{X,Y}(f)$.

It immediately follows from Proposition 5.4.37 that $[\![-]\!]^{Tr}_{\mathbb{P}}$ is a traced CDMU functor that preserves generators. Now we are ready to present some fixed-point semantics for definite logic programs.

Consequence operators and Herbrand semantics

As already introduced in Section 2.1.1, 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 canonical models are precisely the least fixed points of the immediate consequence operators, which exist because the operators are monotonic on the complete lattice $\mathcal{P}(At)$, or equivalently, $\mathbb{B}_{At} = \prod_{A \in At} \mathbb{B}_A$. This suggests that one can express the least Herbrand models $\mathcal{M}^H$ (*cf.* Definition 2.1.7) simply as immediate consequence operators plus 'feedback wires'. We first deal with consequence operators, by which Herbrand models can be defined straightforwardly.

Definition 4.2.15. The *consequence diagram* $C_{\mathbb{L}}$ is a $\mathsf{Syn}^{tr}_{\mathbb{L}}$-morphism defined as follows, where the box is the immediate consequence diagram $T_{\mathbb{L}}$ in (4.1):



(4.4)

In traditional categorical terms, $C_{\mathbb{L}}$ is

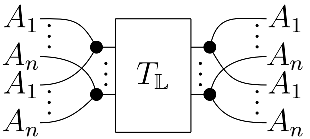
$$Tr^{[A_1,\dots,A_n]}_{[A_1,\dots,A_n],[A_1,\dots,A_n]}((\sqsupset\!\!\bullet\!\!-_{A_1}\otimes\cdots\otimes\sqsupset\!\!\bullet\!\!-_{A_n})\ ;\ T_{\mathbb{L}}\ ;\ (-\!\!\bullet\!\!\sqsubset_{A_1}\otimes\cdots\otimes-\!\!\bullet\!\!\sqsubset_{A_n}))$$

and in this form it is not easy to comprehend. This again emphasises the advantage of diagramamtic language which is succinct and intuitive, yet mathematically strict. We include this following statement which seems to be a repetition of Proposition 4.2.8, yet stated instead for the new model $[\![-]\!]^{Tr}_{\mathbb{P}}$ that we use here to interpret diagrams with feedbacks.

Lemma 4.2.16. $[\![T_{\mathbb{L}}]\!]^{Tr}_{\mathbb{P}} = \mathbf{T}_{\mathbb{P}}$.

Proof. Note that $T_{\mathbb{L}}$ is trace-free, and $[\![\cdot]\!]^{Tr}_{\mathbb{P}}$ and $[\![\cdot]\!]_{\mathbb{P}}$ coincide on the trace-free fragment of $\mathsf{Syn}^{tr}_{\mathbb{L}}$, the claim follows immediately from Proposition 4.2.8. □

Proposition 4.2.17. *The string diagram $C_{\mathbb{L}}$ in* (4.4) *expresses the consequence operator of* $\mathbb{P}$*, in the sense that* $[\![C_{\mathbb{L}}]\!]^{Tr}_{\mathbb{P}} = \mathbf{C}_{\mathbb{P}}$.

Proof. For readability, we denote the diagram  as g, so the consequence diagram $C_{\mathbb{L}}$ is $Tr(g)$ (where we omit the obvious sub/superscripts for Tr). Then for arbitrary $(a_1,\dots,a_n)\in\mathbb{B}_{A_1}\times\cdots\times\mathbb{B}_{A_n}$,

$$\begin{aligned}
[\![C_{\mathbb{L}}]\!]^{Tr}_{\mathbb{P}}(a_1,\cdots,a_n) &= \left[\!\!\left[Tr^{[A_1,\dots,A_n]}_{[A_1,\dots,A_n],[A_1,\dots,A_n]}(g)\right]\!\!\right]^{Tr}_{\mathbb{P}}(a_1,\dots,a_n)\\
&= Tr^{\mathbb{B}_{A_1}\times\cdots\mathbb{B}_{A_n}}_{\mathbb{B}_{A_1}\times\cdots\mathbb{B}_{A_n},\mathbb{B}_{A_1}\times\cdots\mathbb{B}_{A_n}}([\![g]\!]^{Tr}_{\mathbb{P}})(a_1,\dots,a_n)\\
&= \pi_{n+1,\dots,2n}\circ[\![g]\!]^{Tr}_{\mathbb{P}}(u_1,\dots,u_n,a_1,\dots,a_n)
\end{aligned}$$

where $(u_1,\dots,u_n)\in\mathbb{B}_{A_1}\times\cdots\times\mathbb{B}_{A_n}$ is the least fixed point of $\lambda\mathbf{u}.\pi_{1,\dots,n}\circ[\![g]\!]^{Tr}_{\mathbb{P}}(\mathbf{u},\mathbf{a})$. Since

$$\begin{aligned}
\mu\mathbf{u}.\pi_{1,\dots,n}\circ[\![g]\!]^{Tr}_{\mathbb{P}}(\mathbf{u},\mathbf{a}) &= \mu\mathbf{u}.\pi_{1,\dots,n}\circ\left[\!\!\left[-\!\!\bullet\!\!\sqsubset_{[A_1,\dots,A_n]}\circ T_{\mathbb{L}}\circ\sqsupset\!\!\bullet\!\!-_{[A_1,\dots,A_n]}\right]\!\!\right]^{Tr}_{\mathbb{P}}(\mathbf{u},\mathbf{a})\\
&= \mu\mathbf{u}.\pi_{1,\dots,n}\circ\left[\!\!\left[-\!\!\bullet\!\!\sqsubset_{[A_1,\dots,A_n]}\right]\!\!\right]^{Tr}_{\mathbb{P}}\circ[\![T_{\mathbb{L}}]\!]^{Tr}_{\mathbb{P}}(u_1\vee a_1,\dots,u_n\vee a_n)\\
&= \mu u_1,\dots,u_n.\,[\![T_{\mathbb{L}}]\!]^{Tr}_{\mathbb{P}}(u_1\vee a_1,\dots,u_n\vee a_n)\\
&= \mu u_1,\dots,u_n.\mathbf{T}_{\mathbb{P}}(u_1\vee a_1,\dots,u_n\vee a_n)\\
&= \mathbf{C}_{\mathbb{P}}(a_1,\dots,a_n)
\end{aligned}$$

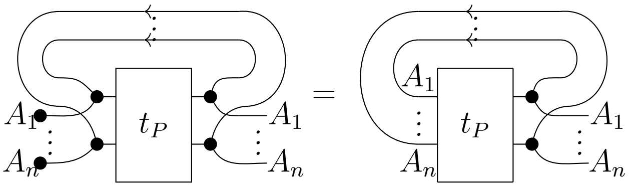
we have

$$\begin{aligned}
\llbracket C_{\mathbb{L}} \rrbracket_{\mathbb{P}}^{\mathit{Tr}} (a_1, \ldots, a_n) &= \pi_{n+1,\ldots,2n} \circ \llbracket g \rrbracket_{\mathbb{P}}^{\mathit{Tr}} (u_1, \ldots, u_n, a_1, \ldots, a_n) \\
&= \pi_{n+1,\ldots,2n} \circ \llbracket g \rrbracket_{\mathbb{P}}^{\mathit{Tr}} (\mathbf{C}_{\mathbb{P}}(a_1, \ldots, a_n), a_1, \ldots, a_n) \\
&= \pi_{n+1,\ldots,2n}(\mathbf{C}_{\mathbb{P}}(a_1, \ldots, a_n), \mathbf{C}_{\mathbb{P}}(a_1, \ldots, a_n)) \\
&= \mathbf{C}_{\mathbb{P}}(a_1, \ldots, a_n)
\end{aligned}$$

Therefore $\llbracket C_{\mathbb{P}} \rrbracket_{\mathbb{L}}^{\mathit{Tr}} = \mathbf{C}_{\mathbb{P}}$. □

The Herbrand semantics, or least model semantics, of definite logic programs, is then represented diagrammatically using the consequence diagrams together with ●— which stands for for input value 0.

Corollary 4.2.18. *The following string diagram $H_{\mathbb{L}}$ expresses the least Herbrand model of $\mathbb{P}$, in the sense that $\llbracket H_{\mathbb{L}} \rrbracket_{\mathbb{P}}^{\mathit{Tr}} = \mathcal{M}_{\mathbb{P}}^{H}$.*



(4.5)

Proof. Note that $\llbracket$●—$\rrbracket_{\mathbb{P}}^{\mathit{Tr}}$ denotes 0. Then

$$\begin{aligned}
\llbracket H_{\mathbb{L}} \rrbracket_{\mathbb{P}}^{\mathit{Tr}} &= \llbracket (\bullet\!\!-_{A_1} \otimes \cdots \otimes \bullet\!\!-_{A_n}) \,;\, C_{\mathbb{L}} \rrbracket_{\mathbb{P}}^{\mathit{Tr}} \\
&= \llbracket C_{\mathbb{L}} \rrbracket_{\mathbb{P}}^{\mathit{Tr}} \circ \llbracket \bullet\!\!-_{A_1} \otimes \cdots \otimes \bullet\!\!-_{A_n} \rrbracket_{\mathbb{P}}^{\mathit{Tr}} \\
&= \mathbf{C}_{\mathbb{P}}(0_{A_1}, \ldots, 0_{A_n}) && \text{(Proposition 4.2.17)} \\
&= \mathcal{M}_{\mathbb{P}}^{H} && \text{(Definition 2.1.7)}
\end{aligned}$$

□

4.3 Probabilistic logic programming

In this section we turn to probabilistic logic programming (PLP). The structure of this section is as follows. We first present the functorial semantics of PLP in Section 4.3.1. Next we discuss a pictorial representation of distribution semantics, in Section Section 4.3.2. We justify the diagrammatic representation of PLP syntax as string diagrams by comparison with *Bayesian networks* (BNs): we explore this in Section 4.3.3, where we show the equivalence between Boolean-valued BNs and acyclic PLP programs, and then illustrate their relationship with functors between the underlying 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. This formalises the intuition that the difference between the two formalisms is only at the semantics level. Second, we present a formal correspondence between (acyclic) PLP programs and Boolean-valued BNs using only transformation at the syntactic level. This reveals that PLP and BN differ in their syntactic structures, and their mutual transformation lies in the syntactic region.

4.3.1 Functorial Semantics of PLP

We start with the functorial semantics of PLP. Throughout this section we fix a PLP program $\mathbb{P}$ over At, and denote its underlying inference structure $\mathit{inf}(\mathbb{P})$ as $\mathbb{L}$. Our goal is to provide for PLP a functorial semantics characterisation analogous to the one for logic programs (Proposition 4.2.5). As mentioned above, we will use the same syntactic category as that for LP (see Section 4.1). As for the semantic category, one needs to switch from Boolean-valued functions to their probabilistic counterpart.

Definition 4.3.1. The category $\mathbf{Stoch}(\mathbb{B})$ has the following structure:

- The objects are the same as that for $\mathsf{Set}(\mathbb{B})$, namely finite products of two-element Boolean algebras.
- Morphisms $\mathbb{B}_1 \times \cdots \times \mathbb{B}_k \to\!\!\!\!\triangleright\ \mathbb{B}_1 \times \cdots \times \mathbb{B}_m$ are functions of the form $f\colon \mathbb{B}_1 \times \cdots \times \mathbb{B}_k \to \mathcal{D}(\mathbb{B}_1 \times \cdots \times \mathbb{B}_m)$, where $\mathcal{D}(\mathbb{B}_1 \times \cdots \times \mathbb{B}_m)$ is the set of probability distributions on $\mathbb{B}_1 \times \cdots \times \mathbb{B}_m$.
- The identity morphism $id_{\mathbb{B}}$ is the Dirac distribution on $1 \in \mathbb{B}$, namely $id_{\mathbb{B}} = 1|1\rangle$;
- Given morphisms $f\colon X \to \mathcal{D}(Y)$ and $g\colon Y \to \mathcal{D}(Z)$, their composition $g \circ f\colon X \to \mathcal{D}(Z)$ assigns to each $x \in X$ a distribution $\sum_{z\in Z} \left(\sum_{y\in Y} f(x)(y) \cdot g(y)(z)\right) |z\rangle$.

An alternative description of $\mathbf{Stoch}(\mathbb{B})$ is that it is the full subcategory of the category of stochastic matrices $\mathbf{Stoch}$ whose objects are restricted to those of $\mathsf{Set}(\mathbb{B})$. Since $\mathbf{Stoch}$ is exactly the Kleisli category for the distribution monad $\mathcal{D}$, we may regard composition in $\mathbf{Stoch}(\mathbb{B})$ simply as Kleisli composition, and use $\to\!\!\!\!\triangleright$ for arrows in $\mathbf{Stoch}(\mathbb{B})$ to distinguish it from their function counterpart. For example, we use $\mathbb{B} \to\!\!\!\!\triangleright\ \mathbb{B}$ and $\mathbb{B} \to \mathcal{D}(\mathbb{B})$ interchangeably for the types of $\mathbf{Stoch}(\mathbb{B})$-morphisms. As before, we verify that $\mathbf{Stoch}(\mathbb{B})$ has enough structure to interpret the syntactic categories.

Lemma 4.3.2. *$\mathbf{Stoch}(\mathbb{B})$ is a CDMU category.*

Proof. First of all, since $\mathbf{Stoch}(\mathbb{B})$ is a full subcategory of the SMC $\mathbf{Stoch}$, $\langle \mathbf{Stoch}(\mathbb{B}), \times, \mathbf{1}\rangle$

forms a SMC as well [JKZ19]. Moreover, we define the CDMU structure on $\mathbb{B}$ as follows:

$$\begin{array}{ccccc@{\qquad}ccccc}
\nabla & : & \mathbb{B} & \rightarrowtriangle & \mathbb{B}\times\mathbb{B} & \epsilon & : & \mathbb{B} & \rightarrowtriangle & \mathbf{1} \\
 & & x & \mapsto & 1|(x,x)\rangle & & & x & \mapsto & 1|\bullet\rangle \\
\Delta & : & \mathbb{B}\times\mathbb{B} & \rightarrowtriangle & \mathbb{B} & \eta & : & \mathbf{1} & \rightarrowtriangle & \mathbb{B} \\
 & & (x_1,x_2) & \mapsto & 1|x_1\vee x_2\rangle & & & \bullet & \mapsto & 1|1\rangle
\end{array}$$

We only verify the CDMU equations for the monoid structure, and that for the comonoid structure can be found in [JKZ19], for the CD category **Stoch**. Given arbitrary $x, y, z \in \mathbb{B}$,

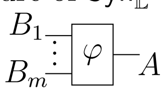
- Unital: $((id \otimes \eta) \ ; \Delta)(x) = 1|x \vee 0\rangle = 1|x\rangle = id(x)$. Similarly $(\eta \otimes id) \ ; \Delta = id$.
- Associative: $((\Delta \otimes id) \ ; \Delta)(x, y, z) = 1|x \vee y \vee z\rangle = ((id \otimes \Delta) \ ; \Delta)(x, y, z)$.
- Commutative: $(\gamma \ ; \Delta)(x, y) = 1|x \vee y\rangle = \Delta(x, y)$.

The CDMU structure on arbitrary objects of the form $\mathbb{B}_1 \times \cdots \times \mathbb{B}_k$ are defined pointwise, and it follows immediately that the structure satisfies CDMU equations. □

Now we are ready to formalise the functorial semantics of PLP. Similar to the case of LP, we will present PLP as certain structure-preserving functors, and the other way around, such that they form the 1-1 correspondence between PLP programs and functors. The idea is that the probabilistic feature of PLP can be fully captured by the images of its associated functors in the category **Stoch**($\mathbb{B}$).

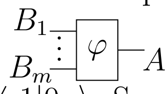
On one hand, we start with a PLP $\mathbb{P}$ with *inf*($\mathbb{P}$) = $\mathbb{L}$. The associated functor will map every generating morphism ψ of $\mathsf{Syn}_{\mathbb{L}}$ to the Markov kernel it represents.

Definition 4.3.3. The functor $[\![-]\!]_{\mathbb{P}} : \mathsf{Syn}_{\mathbb{L}} \to \mathbf{Stoch}(\mathbb{B})$ is defined as follows:

- on objects, $[\![-]\!]_{\mathbb{P}}$ maps $A \in At$ to $\mathbb{B}_A$;
- on morphisms, $[\![-]\!]_{\mathbb{P}}$ maps the CDMU structure of $\mathsf{Syn}_{\mathbb{L}}$ to that of **Stoch**($\mathbb{B}$) as shown in Lemma 4.3.2; for each generating morphism  in $\mathsf{Syn}_{\mathbb{L}}$,

$$[\![\varphi]\!]_{\mathbb{P}} : u \mapsto \begin{cases} p|1_A\rangle + (1-p)|0_A\rangle & \exists\psi \text{ such that } \tau^{pr}(\psi) = \varphi, u \vDash \mathsf{body}(\psi), \text{ and } lab(\psi) = p \\ 1|0_A\rangle & \text{else} \end{cases}$$

On the other hand, a generator-preserving CDMU functor $\mathcal{F}\colon \mathsf{Syn}_{\mathbb{L}} \to \mathbf{Stoch}(\mathbb{B})$ determines a PLP program whose underlying inference structure is $\mathbb{L}$, and its clauses capture exactly the $\mathcal{F}$-image of the generating string diagram of $\mathsf{Syn}_{\mathbb{L}}$.

Definition 4.3.4. The PLP program $Prog_{\mathcal{F}}$ consists of all φ_v defined as below. For each generatig morphism  of $\mathsf{Syn}_{\mathbb{L}}$, let $\{v_1, \ldots, v_s\}$ be the set of all $v \in \mathbb{B}_{B_1} \times \cdots \times \mathbb{B}_{B_m}$ satisfying $\mathcal{F}(\varphi)(v) \neq 1|0_A\rangle$. Suppose:

1. $v(B_{i_1}) = 1_{B_{i_1}}, \dots, v(B_{i_k}) = 1_{B_{i_k}}$ and $v(B_{j_1}) = 0_{B_{j_1}}, \dots, v(B_{j_\ell}) = 0_{B_{j_\ell}}$ exhaust all $B_1, \dots, B_m$;
2. $\mathcal{F}(\varphi)(v) = p|1_A\rangle + (1-p)|0_A\rangle$,

then φ_v is $p :: A \leftarrow B_{i_1}, \dots, B_{i_k}, \neg B_{j_1}, \dots, \neg B_{j_\ell}$.

Example 4.3.5. We exemplify the $\mathit{Prog}_{(\cdot)}$ construction via a toy example. Consider the atom set $At = \{A, B, C\}$ and the definite progred3am $\mathbb{L}$ consisting of two clause φ_1 and φ_2 of the form $A \leftarrow B, C$ and $A \leftarrow B$, respectively. Suppose further that the functor $\mathcal{G}\colon \mathsf{Syn}_{\mathbb{L}} \to \mathbf{Stoch}(\mathbb{B})$ maps the generating morphism φ_1 to the function $\mathcal{G}(\varphi_1)\colon \mathbb{B}_B \times \mathbb{B}_C \to \mathcal{D}(\mathbb{B}_A)$ such that:

$$(0_B, 0_C) \mapsto 1|0_A\rangle \quad (0_B, 1_C) \mapsto 0.3|0_A\rangle{+}0.7|1_A\rangle \quad (1_B, 0_C) \mapsto 1|0_A\rangle \quad (0_B, 0_C) \mapsto 0.5|0_A\rangle{+}0.5|1_A\rangle$$

and maps the generating morhpism φ_2 to the function $\mathcal{G}(\varphi_2)\colon \mathbb{B}_B \to \mathcal{D}(\mathbb{B}_A)$ such that:

$$0_B \mapsto 1|0_A\rangle \qquad 1_B \mapsto 0.9|0_A\rangle + 0.1|1_A\rangle$$

Then the induced program $\mathit{Prog}_{\mathcal{G}}$ consists of the following three clauses:

$$0.7 :: A \leftarrow \neg B, C \qquad 0.5 :: A \leftarrow \neg B, \neg C \qquad 0.1 :: A \leftarrow B$$

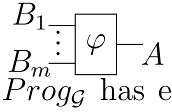
Note that, for instance, $0_B \mapsto 1|0_A\rangle$ does not give rise to a clause, because the clause that corresponds to this semantics would be $0 :: A \leftarrow \neg B$, which is not a valid clause for PLP as its existence does not affect the semantics of the program at all.

Indeed these two directions establish the functorial semantics of PLP.

Proposition 4.3.6. *There is a 1-1 correspondence between PLP programs based on $\mathbb{L}$ and generator-preserving CDMU functors $\mathsf{Syn}_{\mathbb{L}} \to \mathbf{Stoch}(\mathbb{B})$.*

Proof. We show that the aforementioned two constructions $[\![-]\!]_{(\cdot)}$ and $\mathit{Prog}_{(\cdot)}$ are inverse to each other.

On one hand, we start 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 $\tau^{pr}(\psi) = \varphi$. This clause ψ determines the behaviour of the Markov kernel $[\![\varphi]\!]_{\mathbb{P}} : \mathbb{B}_{B_{i_1}} \times \cdots \times \mathbb{B}_{B_{i_k}} \times \mathbb{B}_{B_{j_1}} \times \cdots \times \mathbb{B}_{B_{j_\ell}}$ at the unique state v satisfying $v \vDash \mathsf{body}(\psi)$, such that $[\![\varphi]\!]_{\mathbb{P}}(v) = p|A\rangle + (1-p)|\neg A\rangle$. This defines a unique clause in the program $\mathit{Prog}_{[\![-]\!]_{\mathbb{P}}}$, which is exactly ψ. Moreover, every clause in $\mathit{Prog}_{[\![-]\!]_{\mathbb{P}}}$ is generated in this way: for a clause φ_v to appear in $\mathit{Prog}_{[\![-]\!]_{\mathbb{P}}}$, it must be the case that $[\![\varphi]\!]_{\mathbb{P}}(v) \neq 1|0_A\rangle$, thus there is a clause ψ from the original program $\mathbb{P}$ that determines $[\![\varphi]\!]_{\mathbb{P}}(v)$ is supported.

On the other hand, starting from a generator-preserving CDMU functor $\mathcal{G}$, for each generating morphism  , suppose the set $V^+ = \{v \in \mathbb{B}_{B_1} \times \cdots \times \mathbb{B}_{B_m} \mid \mathcal{G}(\varphi)(v) \neq 1|0_A\rangle\}$ has size s, then $\mathit{Prog}_{\mathcal{G}}$ has exactly s-many clause whose inference structure is φ, say φ_v for each v. Indeed φ_v satisfies that

- $\mathsf{head}(\varphi_v) = A$;
- $B_i \in \mathsf{body}(\varphi_v)$ iff $v(B_i) = 1$;
- $\neg B_j \in \mathsf{body}(\varphi_v)$ iff $v(B_j) = 0$;
- no other literals appear in $\mathsf{body}(\varphi_v)$.

Together these s-many φ_v determine the behaviour of $[\![-]\!]_{Prog_\mathcal{G}}$ on φ, and $[\![\varphi]\!]_{Prog_\mathcal{G}}$ maps $u \in \mathbb{B}_{B_1} \times \cdots \times \mathbb{B}_{B_m}$ to $1|0_A\rangle$ if u does not match the body any of those φ_v; it maps u to $p|1_A\rangle + (1-p)|0_A\rangle$ if it matches the body of some φ_v such that $lab(\varphi_v) = p$. This means that $[\![\varphi]\!]_{Prog_\mathcal{G}} = \mathcal{G}(\varphi)$. Thus $[\![-]\!]_{Prog_\mathcal{G}} = \mathcal{G}$. □

4.3.2 Distribution semantics in string diagrams

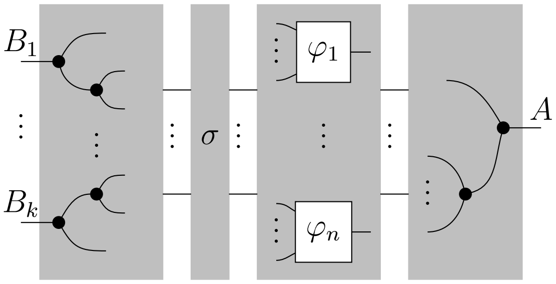
As an immediate application of the functorial semantics of PLP, we now study diagrammatic representations of PLP semantics, in a similar flavour to that for LP. We restrict ourselves to acyclic programs, which have a straightforward distribution semantics (Definition 2.1.11). This enables us to stick with the simpler diagrammatic language $\mathsf{Syn}_\mathbb{L}$, instead to moving to the diagrammatic language with traces $\mathsf{Syn}^{tr}_\mathbb{L}$. Let us formalise the intuitive notion of *acyclic* program first.

Definition 4.3.7. A definite logic program $\mathbb{L}$ is *acyclic* if 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 $\mathsf{head}(\varphi_0) \in \mathsf{body}(\varphi_m)$. A (classical, probabilistic, weighted) logic program $\mathbb{P}$ is acyclic if its underlying inference striucture $\mathit{inf}(\mathbb{P})$ is acyclic.

In words, program $\mathbb{L}$ contains a cycle if there is some reduction chain from heads to bodies of clauses, such that some goal is eventually reduced to itself. Thus a trace-free diagrammatic language suffices for acyclic programs because there is no feedback needed to depict the derivation of the atoms of interest, which expresses the semantics of these atoms.

The goal of this subsection is to diagrammatically express the probability distribution of a set of atoms under the standard distribution semantics. The idea is to construct a string diagram $f_{\bar{A}} \colon [\,] \to [A_1, \ldots, A_k]$ which exhausts all possible derivations of $\{A_1, \ldots, A_k\}$ in $\mathbb{P}$, and its interpretation $[\![f_{\bar{A}}]\!]_\mathbb{P}$ returns exactly the desired distribution semantics of $\bar{A}$ in $\mathbb{P}$. As a preliminary, we introduce below the notion of 'A-component'. Intuitively the A-component collects all the clauses with heads A, and 'bundle' them into a string diagram whose output wire is A, and whose input wires does not have duplicate of atoms. We separate out this concept because of its importance in both the diagrammatic distribution semantics and the comparison of PLP with the diagrammatic representation of Bayesian networks in the next subsection.

Definition 4.3.8. The A-*component* in $\mathbb{L}$ is a string diagram $comp_A\colon [B_1,\dots,B_k] \to [A]$ in $\mathsf{Syn}_{\mathbb{L}}$ below, where:



1. $B_1,\dots,B_k$ are all the atoms B satisfying that there exists $\varphi \in \mathbb{L}$ such that $\mathsf{head}(\varphi) = A$ and $B \in \mathsf{body}(\varphi)$.

2. In the first block, for each atom B_i, 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 $B \in \mathsf{body}(\varphi)$.

3. The third block contains parallel string diagrams all each $\mathbb{L}$-clause $\varphi_1,\dots,\varphi_n$ with head A.

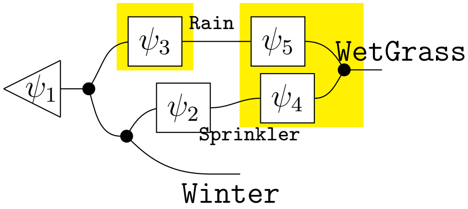
4. In the fourth block, there are n-many cocopies of A (via ⊐●—), where n is the number of $\mathbb{L}$-clauses with head A.

5. The second block (marked with σ) contains suitably many swapping morphisms ⊃⊂ to match each copy of B_i in the first block to an input wire B_i for some φ in the third block.

Note that the above definition covers the corner case where $n = 0$, namely the case where there is no $\mathbb{L}$-clause with head A. In this situation, the definition says that we make 0 copy of A, and $comp_A$ simplifies to ●—$_A$. This matches our intuition that, if there is no clause with head A, then A is not derivable thus should have truth value 0, expressed by ●—.

To construct the diagram expressing the joint distribution of a set of atoms $\bar{A} := \{A_1,\dots,A_k\}$, we basically need to exhaust all the possible derivations in $\mathbb{P}$ of each of these A_i, and then compose them in a suitable manner. The key step of exhausting all the possible derivations is to form A-components. We illustrate this idea using a simple example before diving into the formal construction — the later is not hard but cumbersome.

Example 4.3.9. Recall the program $\mathbb{P}_{\mathrm{wet}}$ from Example 4.1.3. To consider the joint distribution of the set of atoms $\{\mathtt{WetGrass}, \mathtt{Winter}\}$ in $\mathbb{P}_{\mathrm{wet}}$, intuitively we want to exhaust all possible derivations of `WetGrass` and `Winter` in $\mathbb{P}_{\mathrm{wet}}$ from scratch, such that the only 'open' wires are `WetGrass` and `Winter`:



In particular, the highlighted parts in the above diagram are $comp_{\mathtt{Rain}}$ and $comp_{\mathtt{WetGrass}}$, respectively.

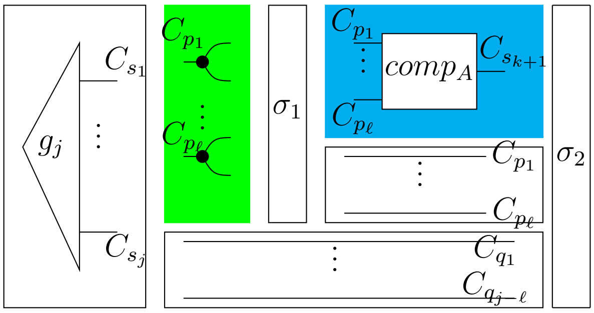
We need the following handy notion of ancestors of atoms in a program to facilitate the construction of distribution diagrams.

Definition 4.3.10. An atom B is an *ancestor* of atom A in $\mathbb{L}$ if there exists a chain of $\mathbb{L}$-clauses $\varphi_1, \ldots, \varphi_k$ such that $B \in \mathsf{body}(\varphi_1), \mathsf{head}(\varphi_i) \in \mathsf{body}(\varphi_{i+1})$ for all $i = 1, \ldots, k-1$, and $\mathsf{head}(\varphi_k) = A$. We denote the set of all ancestors of A in $\mathbb{L}$ as $\mathsf{Ancestor}_{\mathbb{L}}(A)$, and the union of ancestors $\bigcup_{A \in \mathcal{A}} \mathsf{Ancestor}_{\mathbb{L}}(A)$ as $\mathsf{Ancestor}_{\mathbb{L}}(\mathcal{A})$.

For example, in $\mathbb{P}_{\text{wet}}$ (Example 4.3.9), the atom `Sprinkler` an ancestor of `WetGrass`, but not an ancestor of `Rain`.

Definition 4.3.11. Let $\mathcal{B} = \{B_1, \ldots, B_k\}$ be a subset of At. The *PLP distribution diagram* $f_{\mathcal{B}} \colon [\,] \to [B_1, \ldots, B_k]$ is defined using the following construction of a sequence of diagrams $\{g_i\}_{i=1}^m$, where we assume $\{B_1, \ldots, B_k\} \cup \mathsf{Ancestor}_{\mathbb{L}}(\{B_1, \ldots, B_k\}) = \{C_1, \ldots, C_m\}$:

- g_0 is the empty diagram;
- Suppose g_j is defined for some $j < m$, and is of type $[\,] \to [C_{s_1}, \ldots, C_{s_j}]$, then g_{j+1} is defined as the following string diagram:



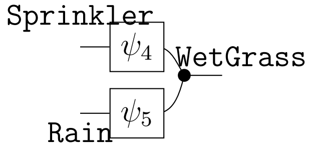
In particular, we pick an $C_{s_{j+1}} \in \{C_1, \ldots, C_m\} \setminus \{C_{s_1}, \ldots, C_{s_j}\}$ such that all atoms in $\mathsf{dom}(comp_{C_{s_{j+1}}})$ already appear in $C_{s_1}, \ldots, C_{s_j}$, say $comp_{C_{s_{j+1}}} : [C_{p_1}, \ldots, C_{p_\ell}] \to [C_{s_{j+1}}]$ with $\{C_{p_1}, \ldots, C_{p_\ell}\} \subseteq \{C_{s_1}, \ldots, C_{s_j}\}$. In the green block we make two copies for each atom in $\{C_{p_1}, \ldots, C_{p_\ell}\}$ from the codomain of g_j in the leftmost block. σ_1 consists of suitably many $\supset\!\!\subset$ to match one of the copies of each C_{p_i} to the left port of $comp_A$ in the blue block. σ_2 again contains $\supset\!\!\subset$ that make the codomain $[C_{s_1}, \ldots, C_{s_{j+1}}]$.

Finally, the distribution diagram is defined as $f_{\mathcal{B}} = g_m \,;\, (h_1 \otimes \cdots \otimes h_m)$, where each $h_i = \text{——}_{C_i}$ if $C_i \in \mathcal{B}$, and $h_i = \text{—}\!\bullet_{C_i}$ if $C_i \notin \mathcal{B}$.

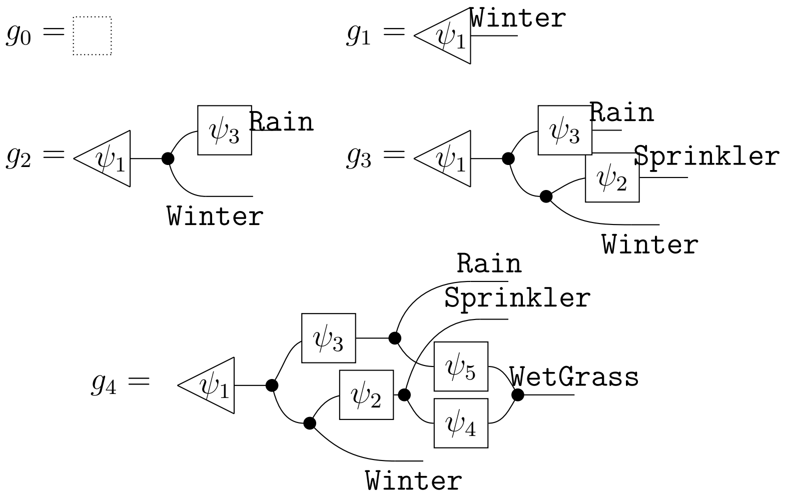
In words, one first construct a diagram g_m whose wires on the right port represent exactly every $B \in \mathcal{B}$ and their ancestors, and $f_{\mathcal{B}}$ is obtained by dropping all the atoms in $\mathsf{Ancestor}_{\mathbb{L}}(\mathcal{B}) \backslash \mathcal{B}$ using $\text{—}\!\bullet$.

Example 4.3.12. The construction of $f_{\mathcal{B}}$ can be exemplified by the running example Example 4.1.3. The set of variables of interest is $\mathcal{B} = \{\texttt{Winter}, \texttt{WetGrass}\}$. Their ancestor set

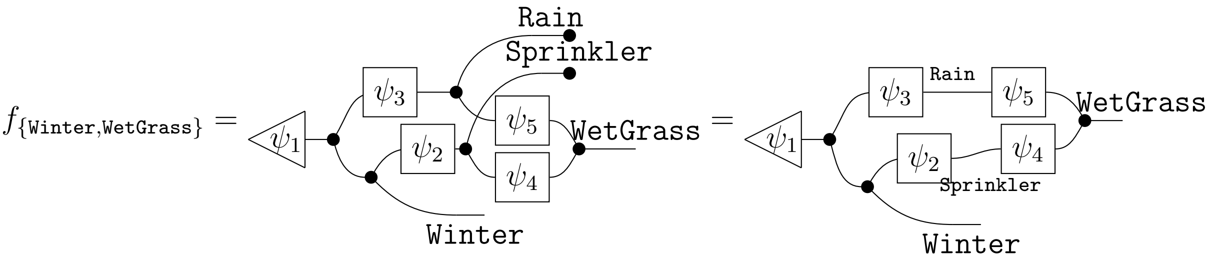
$\mathsf{Ancestor}_{\mathbb{L}_{\mathrm{wet}}}(\mathcal{B})$ is $\{\texttt{Winter}, \texttt{Sprinkler}, \texttt{Rain}\}$. For instance, the $\texttt{WetGrass}$-component is:



Now we are ready to construct the series of diagrams $\{g_i\}_{i=0}^{4}$ (note that 4 is the size of $\mathcal{B} \cup \mathsf{Ancestor}_{\mathbb{L}_{\mathrm{wet}}}(\mathcal{B})$) as follows:



Finally, by discarding the atoms $\texttt{Rain}, \texttt{Sprinkler}$ in $\mathsf{Ancestor}_{\mathbb{L}_{\mathrm{wet}}}(\mathcal{B}) \backslash \mathcal{B}$, we obtain $f_{\{\texttt{Winter},\texttt{WetGrass}\}} = g_4 \,;\, (\text{—●}_{\texttt{Rain}} \otimes \text{—●}_{\texttt{Sprinkler}} \otimes id_{\texttt{WetGrass}} \otimes id_{\texttt{Winter}})$, or diagrammatically,



In particular, note that $\texttt{SlipperyRoad} \in At_{\mathrm{wet}}$ did not appear in the construction of $f_{\{\texttt{Winter},\texttt{WetGrass}\}}$, because it appears in neither $\{\texttt{Winter}, \texttt{WetGrass}\}$ nor $\mathsf{Ancestor}_{\mathbb{P}_{\mathrm{wet}}}(\{\texttt{Winter}, \texttt{WetGrass}\})$.

The distribution diagram calculates exactly the distribution of the atoms of interest, under the interpretation $[\![-]\!]_{\mathbb{P}}$.

Proposition 4.3.13. *$f_{\mathcal{B}}$ expresses the distribution semantics: $[\![f_{\mathcal{B}}]\!]_{\mathbb{P}} = \delta(\mathcal{B})$.*

Proof. Let us follow the same assumption and notation in Definition 4.3.11. Moreover, without loss of generality we assume that $\{C_1, \ldots, C_m\} \backslash \{B_1, \ldots, B_k\} = \{D_1, \ldots, D_{m-k}\}$. We prove the following result for the series of diagrams $\{g_i\}_{i=0}^{m}$: for each g_i, say of type $[\,] \to [C_{s_1}, \ldots, C_{s_i}]$, $[\![g_i]\!]_{\mathbb{P}_{\mathrm{wet}}} = \delta(\{C_{s_1}, \ldots, C_{s_i}\})$. Suppose this is true, then the original statement holds since

$$[\![f_{\mathcal{B}}]\!]_{\mathbb{P}} = \left[\!\!\left[g_m \,;\, (\text{——}_{B_1} \otimes \cdots \otimes \text{——}_{B_k} \otimes \text{—●}_{D_1} \otimes \cdots \otimes \text{—●}_{D_{m-k}}) \right]\!\!\right]_{\mathbb{P}}$$

$$
\begin{aligned}
&= \delta(\{B_1, \ldots, B_k, D_1, \ldots, D_{m-k}\}) \mathbin{;} \left[\!\left[\text{——}_{B_1} \otimes \cdots \otimes \text{——}_{B_k} \otimes \text{—}\!\bullet_{D_1} \otimes \cdots \otimes \text{—}\!\bullet_{D_{m-k}} \right]\!\right]_{\mathbb{P}} \\
&= \delta(\{B_1, \ldots, B_k\})
\end{aligned}
$$

where the last equation uses the fact that the semantic category $\mathbf{Stoch}(\mathbb{B})$ is a Markov category in which $\text{—}\!\bullet$ is natural.

So now we focus on the statement about g_i. We prove by induction on $i = 0, \ldots, m$. For $i = 0$ the statement is trivial. For $i = 1$, $g_1 \colon [\,] \to [C_{s_1}]$. By the choice of C_{s_1}, the domain of $comp_{C_{s_1}}$ is already contained in $\varnothing$, namely $comp_{C_{s_1}}$ has type $\varnothing \to [C_{s_1}]$. In other words, all clauses in $\mathbb{P}$ with heads C_{s_1} have empty bodies (note that it is possible that there is no clause with head C_{s_1} at all). Then $g_1 = comp_{C_{s_1}}$, and $[\![comp_{C_{s_1}}]\!]_{\mathbb{P}}$ is a distribution in $\mathcal{D}(\mathbb{B}_{C_{s_1}})$, such that $[\![comp_{C_{s_1}}]\!]_{\mathbb{P}}(0_{C_{s_1}}) = \prod_{\mathsf{head}(\psi) = C_{s_1}} lab(\psi)$. This is exactly $\delta(C_{s_1})(0_{C_{s_1}})$ under the distribution semantics, so $[\![g_1]\!]_{\mathbb{P}} = \delta(C_{s_1})$.

For the induction step, suppose $[\![g_i]\!]_{\mathbb{P}} = \delta(C_{s_1}, \ldots, C_{s_i})$ holds (IH) for some $i < m$. We prove the statement for g_{i+1}. Without loss of generality assume further that $comp_{g_{i+1}}$ has domain $[C_{s_1}, \ldots, C_{s_\ell}]$. On one hand,

$$
\begin{aligned}
[\![g_{i+1}]\!]_{\mathbb{P}} &= \left[\!\!\left[g_i \mathbin{;} (\text{—}\!\bullet\!\!\subset_{[C_{s_1},\ldots,C_{s_\ell}]} \otimes \text{——}_{[C_{s_{\ell+1}},\ldots,C_{s_i}]}) \mathbin{;} (comp_{C_{s_{i+1}}} \otimes \text{——}_{[C_{s_1},\ldots,C_{s_i}]}) \right]\!\!\right]_{\mathbb{P}} \\
&= [\![g_i]\!]_{\mathbb{P}} \mathbin{;} \left[\!\!\left[(\text{—}\!\bullet\!\!\subset_{[C_{s_1},\ldots,C_{s_\ell}]} \otimes \text{——}_{[C_{s_{\ell+1}},\ldots,C_{s_i}]}) \mathbin{;} (comp_{C_{s_{i+1}}} \otimes \text{——}_{[C_{s_1},\ldots,C_{s_i}]}) \right]\!\!\right]_{\mathbb{P}} \\
&= \delta(C_{s_1}, \ldots, C_{s_i}) \mathbin{;} \left[\!\!\left[(\text{—}\!\bullet\!\!\subset_{[C_{s_1},\ldots,C_{s_\ell}]} \otimes \text{——}_{[C_{s_{\ell+1}},\ldots,C_{s_i}]}) \mathbin{;} (comp_{C_{s_{i+1}}} \otimes \text{——}_{[C_{s_1},\ldots,C_{s_i}]}) \right]\!\!\right]_{\mathbb{P}} \quad \text{(IH)}
\end{aligned}
$$

Thus for each value $(c_{s_1}, \ldots, c_{s_i}, c_{s_{i+1}}) \in \mathbb{B}_{C_{s_1}} \times \cdots \times \mathbb{B}_{C_{s_{i+1}}}$,

$$
\begin{aligned}
[\![g_{i+1}]\!]_{\mathbb{P}}(c_{s_1}, \ldots, c_{s_i}, c_{s_{i+1}}) &= [\![g_i]\!]_{\mathbb{P}}(c_{s_1}, \ldots, c_{s_i}) \cdot \left[\!\!\left[comp_{C_{s_{i+1}}} \right]\!\!\right]_{\mathbb{P}} (c_{s_1}, \ldots, c_{s_\ell})(c_{s_{i+1}}) \\
&= \delta(C_{s_1}, \ldots C_{s_i})(c_{s_1}, \ldots c_{s_i}) \cdot \left[\!\!\left[comp_{C_{s_{i+1}}} \right]\!\!\right]_{\mathbb{P}} (c_{s_1}, \ldots, c_{s_\ell})(c_{s_{i+1}}) \\
&= \delta(C_{s_1}, \ldots C_{s_i}, C_{s_{i+1}})(c_{s_1}, \ldots c_{s_i}, c_{s_{i+1}})
\end{aligned}
$$

Therefore $[\![g_{i+1}]\!]_{\mathbb{P}} = \delta(C_{s_1}, \ldots C_{s_i}, C_{s_{i+1}})$. □

Example 4.3.14. Continuing Example 4.3.12, the distribution diagram $f_{\{\mathtt{WetGrass},\mathtt{Winter}\}}$ has interpretation $[\![f_{\{\mathtt{WetGrass},\mathtt{Winter}\}}]\!]_{\mathbb{P}_{\mathrm{wet}}} =$, which is exactly the distribution $\delta(\{\mathtt{WetGrass}, \mathtt{Winter}\})$ for atoms under $\mathbb{P}_{\mathrm{wet}}$.

4.3.3 Correspondence of PLP and BNs, from a functorial semantics perspective

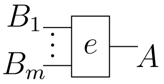
In this section we turn to an application of the functorial semantics of PLP which is of different flavour compared the discussion on LP. We study the relationship between PLP and Bayesian networks (Definition 2.1.13). Beyond their intuitive resemblance, formally every acyclic PLP can be transformed into an equivalent Boolean-valued BN, and vice versa (see e.g. Meert [MSB08]). Let us first provide some intuition of this two-way translation. Starting from a Bayesian network, the structure of its DAG can be described via a definite logic program containing clauses of the form $pa(A) \to A$, and the conditional probability distributions $\Pr(A|B_1, \ldots, B_k)$ can be expressed using (at most) 2^k-many PLP clauses. On the other hand, given an acyclic PLP program $\mathbb{P}$, one can encode its inference structure using a DAG, and the distribution semantics as conditional probability distributions on this DAG.

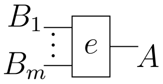
Our goal is formulating this two-way translation between these two formalisms at the functorial level, taking advantage on one side of the functorial semantics of PLP in Subsection **??**, and on the other side of the functorial semantics of Bayesian networks.

4.3.3.1 Overview: functorial semantics of Bayesian networks

Let us first recall the functorial semantics of Bayesian networks, as described in Jacobs et al. [JKZ19] (see also Fong [Fon13]).

Definition 4.3.15. Given a Bayesian network $\mathcal{B} = \langle G, \Pr \rangle$, its syntactic category SynBN_G is the freely generated CD category of string diagrams, such that:

- The generating objects are nodes in G (or equivalently, events in $\mathcal{B}$).
- The generating morpihsms contains one string diagram of the form  for each node A in G, where $B_1, \ldots, B_m$ are all the parents of A in G.

Note that SynBN is only a CD category rather than a CDMU category in that it does not have the monoidal structure ⊐●—, ●—. Its associated functor $[\![-]\!]_{\mathcal{B}} : \mathsf{SynBN}_{\mathcal{B}} \to \mathbf{Stoch}$ maps each generating object A to its value space in $\mathcal{B}$, and maps a generating morphism  to $\Pr(A|B_1, \ldots, B_m)$.

In the aim of comparing BN with PLP, we need a restriction of this characterisation result to *Boolean-valued* Bayesian networks (*cf.* discussion below Definition 2.1.13), which we report below. We note that it is possible to generalise the connection between BN and PLP beyond Bolean-valued ones, with a suitable extension of PLP semantics.

Proposition 4.3.16 ([JKZ19])**.** *There is 1-1 correspondence between Boolean-valued Bayesian networks based on a DAG G and generator-preserving CD functors $\mathsf{Syn}_G^{BN} \to \mathbf{Stoch}(\mathbb{B})$.*

4.3.3.2 From BN to PLP

We begin with the simpler direction, say building a (acyclic) PLP program from a Boolean-valued Bayesian network. We recall the standard construction first, followed by the transformation on the functorial level.

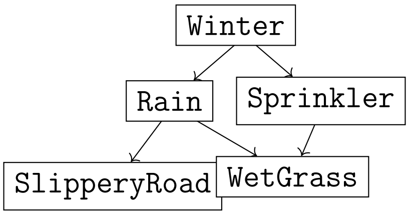
Let us fix a BN $\mathcal{B} = \langle G, \Pr \rangle$ over an event set At, where G is a DAG, and Pr are conditional probability distributions $\Pr(A \mid pa(A))$ for all $A \in At$.

Definition 4.3.17. The *program associated with $\mathcal{B}$* is $\mathit{Prog}(\mathcal{B})$ over At, where for each $A \in At$ and conditional probability distribution $\Pr(A|B_1 = b_1, \ldots, B_k = b_k)$, if $\Pr(A = 1|B_1 = b_1, \ldots, B_k = b_k) = p$ for some $p \neq 0$, then the clause $p :: A \leftarrow L_1, \ldots, L_k$ is in $\mathit{Prog}(\mathcal{B})$, where $L_i = B_i$ if $b_i = 1$, and $L_i = \neg B_i$ if $b_i = 0$, for all $i = 1, \ldots, k$.

The resulting program $\mathit{Prog}(\mathcal{B})$ has the same semantics for all atoms in the following sense.

Proposition 4.3.18. *For arbitrary $C_1, \ldots, C_m \in At$, $\Pr_{\mathcal{B}}(C_1, \ldots, C_m) = \delta_{\mathit{Prog}(\mathcal{B})}(C_1, \ldots, C_m)$.*

Example 4.3.19. We illustrate the $\mathit{Prog}(\cdot)$ construction with the following Bayesian network $\mathcal{B}_{\text{wet}}$ shown as below. The DAG is as follows:



(4.6)

And the conditional probability tables are listed below:

$\Pr(\texttt{Winter})$
0.25

	$\Pr(\texttt{Rain})$
$0_{\texttt{Winter}}$	0.1
$1_{\texttt{Winter}}$	0.6

	$\Pr(\texttt{Sprinkler})$
$0_{\texttt{Winter}}$	0
$1_{\texttt{Winter}}$	0.2

		$\Pr(\texttt{WetGrass})$
$0_{\texttt{Sprinkler}}$	$0_{\texttt{Rain}}$	0
$0_{\texttt{Sprinkler}}$	$1_{\texttt{Rain}}$	0.8
$1_{\texttt{Sprinkler}}$	$0_{\texttt{Rain}}$	0.9
$1_{\texttt{Sprinkler}}$	$1_{\texttt{Rain}}$	0.98

	$\Pr(\texttt{SlipperyRoad})$
$0_{\texttt{Rain}}$	0.1
$1_{\texttt{Rain}}$	0.7

Then $\mathit{Prog}(\mathbb{P}_{\text{wet}})$ consists of the following clauses:

$$\begin{array}{rlllrll}
0.25 :: & \texttt{Winter} & \leftarrow . & \qquad & 0.1 :: & \texttt{Rain} & \leftarrow \neg\texttt{Winter}. \\
0.6 :: & \texttt{Rain} & \leftarrow \texttt{Winter}. & & 0.2 :: & \texttt{Sprinkler} & \leftarrow \texttt{Winter}. \\
0.8 :: & \texttt{WetGrass} & \leftarrow \neg\texttt{Sprinkler}, \texttt{Rain}. & & 0.9 :: & \texttt{WetGrass} & \leftarrow \texttt{Sprinkler}, \neg\texttt{Rain}. \\
0.98 :: & \texttt{WetGrass} & \leftarrow \texttt{Sprinkler}, \texttt{Rain}. & & 0.1 :: & \texttt{SlipperyRoad} & \leftarrow \neg\texttt{Rain} \\
0.7 :: & \texttt{SlipperyRoad} & \leftarrow \texttt{Rain}. & & & &
\end{array}$$

We remark that, this program is different from $\mathbb{P}_{\text{wet}}$ (see Example 4.1.3), in particular they have distinct clauses with head `WetGrass`. Yet these two programs are equivalent in the sense that they have the same distribution semantics.

Now we turn to the functorial semantics perspective. The Bayesian network $\mathcal{B}$ determines a BN model $[\![-]\!]_{\mathcal{B}} : \mathsf{SynBN}_G \to \mathbf{Stoch}(\mathbb{B})$, as in Definition 4.3.15. To derive a PLP model, we first define the syntactic category. Let the definite program $\mathbb{L}$ describing the inference structure be $\mathbb{L} = \{pa(A) \to A \mid A \in At\}$, namely encode the DAG as clauses. The syntactic category is defined as $\mathsf{Syn}_{\mathbb{L}}$. Note that $\mathsf{Syn}_{\mathbb{L}}$ and SynBN_G share the same generating objects and morphisms, and they only differ in that the former is a free CDMU category, while the latter is a free CD category (*i.e.* does not have ⊐•— , •—). Thus the PLP model is simply defined as the CDMU functor $\alpha(\llbracket - \rrbracket_{\mathcal{B}}) \colon \mathsf{Syn}_{\mathbb{L}} \to \mathbf{Stoch}(\mathbb{B})$ obtained by canonically lifting the CD functor $\llbracket - \rrbracket_{\mathcal{B}} : \mathsf{SynBN}_G \to \mathbf{Stoch}(\mathbb{B})$ along the inclusion functor $\iota \colon \mathsf{SynBN}_G \to \mathsf{Syn}_{\mathbb{L}}$ which embeds the CD structure of $\mathsf{Syn}^{BN}_G = \mathsf{freeCD}(\mathsf{V}_G, \Sigma_G)$ into the CDMU structure of $\mathsf{Syn}_{\mathbb{L}} = \mathsf{freeCDMU}(V_G, \Sigma_G)$.

Definition 4.3.20. The PLP model $\alpha(\llbracket - \rrbracket_{\mathcal{B}}) \colon \mathsf{Syn}_{\mathbb{L}} \to \mathbf{Stoch}(\mathbb{B})$ maps each generating morphism φ to $\llbracket \varphi \rrbracket_{\mathcal{B}}$, and maps the CDMU structure in Syn_G to that of $\mathbf{Stoch}(\mathbb{B})$.

The resulting PLP model is exactly that corresponding to the program associated with $\mathcal{B}$ in Definition Definition 4.3.17.

Proposition 4.3.21. $\llbracket - \rrbracket_{Prog(\mathcal{B})} = \alpha(\llbracket - \rrbracket_{\mathcal{B}})$

Proof. Since their domain categories are both the freely generated category $\mathsf{SynPLP}_{\mathbb{L}}$, it suffices to verify that they two behave the same on all generating objects and morphisms. For objects it is obviously true by definition. So we focus on generating morphisms.

For generating morphisms, it is by the definition of $\alpha(\llbracket - \rrbracket_{\mathcal{B}})$ that it maps each generating morphism e of $\mathsf{SynPLP}_{\mathbb{L}}$ (thus of $\mathsf{SynBN}_{\mathcal{B}}$) to $\llbracket e \rrbracket_{\mathcal{B}}$, which is exactly $\llbracket e \rrbracket_{Prog(\mathcal{B})}$. □

The construction of $\alpha(\llbracket - \rrbracket_{\mathcal{B}})$ is simple because it suffices to encode each $Pr(A|pa(A))$ in the Bayesian network by PLP clauses, and the underlying inference structure of the program is exactly encoded by the DAG of the Bayesian network. We shall soon see that this is not the case for the other way around.

4.3.3.3 From PLP to BN

Next we turn to the less straightforward direction, namely encoding an acyclic PLP into a Bayesian network. Let us fix an acyclic PLP program $\mathbb{P}$ over At, with underlying inference structure $\mathbb{L} := \mathit{inf}(\mathbb{P})$.

As for the standard transformation, the associated BN $BN(\mathbb{P})$ is defined as follows.

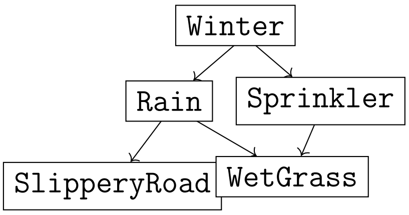
Definition 4.3.22. The *Bayesian network associated with* $\mathbb{P}$ is $BN(\mathbb{P}) = \langle G_{BN(\mathbb{P})}, \mathrm{Pr}_{BN(\mathbb{P})} \rangle$ with the following components:

- In its DAG $G_{BN(\mathbb{P})}$, the vertices are exactly At; a vertex B is a parent of A if and only if there exists a clause φ in $\mathbb{P}$ such that $\mathsf{head}(\varphi) = A$, and B appears in $\mathsf{body}(\varphi)$ (either positively or negatively).

- For its conditional probability distributions $\Pr_{BN(\mathbb{P})}$, given arbitrary A and $pa(A) = \{B_1, \dots, B_k\}$ in $G_{BN(\mathbb{P})}$, $\Pr_{BN(\mathbb{P})}(A = 1 | B_1 = b_1, \dots, B_k = b_k) = 1 - \prod_p (1 - p)$, where p ranges over the probability label of all $\mathbb{P}$-clauses φ such that $\mathsf{head}(\varphi) = A$ and $(B_1 = b_1, \dots, B_k = b_k)$ satisfies $\mathsf{body}(\varphi)$.

Note that in the above definition of $BN(\mathbb{P})$, the structure of DAG $G_{BN(\mathbb{P})}$ is indeed only relies on the inference structure *inf*($\mathbb{P}$). The construction of the conditional distributions $\Pr_{BN(\mathbb{P})}$ deserves some discussion. Intuitively, given an atom A, there can be multiple $\mathbb{P}$-clauses whose heads are A, or in other words, which can be used to prove A. Each clause determines a conditional distribution. Thus to determine the truth value of A one needs to 'bundle together' all those conditional distributions which may have different sets of conditioning events.

Example 4.3.23. We illustrate the construction in Definition 4.3.22 via the running example $\mathbb{P}_{\text{wet}}$ from Example 4.1.3). Its associated Bayesian network $BN(\mathbb{P}_{\text{wet}})$ is shown as below, where on the left is the DAG $G_{BN(\mathbb{P}_{\text{wet}})}$, and on the right is part of the conditional probability table instantiated to `WetGrass`:



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

(4.7)

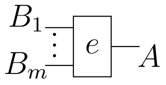
For instance, $\Pr(\texttt{WetGrass} = 1 \mid \texttt{Sprinkler} = 1, \texttt{Rain} = 1) = 1 - (1 - 0.8) \times (1 - 0.9) = 0.98$. Indeed, the resulting Bayesian network $\mathcal{B}_{\text{wet}}$ is that in Example 4.3.23. Since $\mathbb{P}_{\text{wet}}$ is different from $Prog(BN(\mathbb{P}_{\text{wet}}))$, the composition $Prog(\cdot) \circ BN(\cdot)$ is *not* identity.

Before turning to the functorial semantics perspective, we take a closer look at Definition 4.3.22 and try to separate the role of syntax and semantics in this construction. While the construction of the DAG $G_{BN(\mathbb{P})}$ only replies on the underlying inference structure *inf*($\mathbb{P}$) of $\mathbb{P}$, the definition of $\Pr_{BN(\mathbb{P})}$ does not seem to rely solely on the inference structure and thus mingle the syntax and semantics. We claim this view is clarified using the functorial semantics.

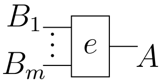
So now let us turn to the functorial semantics perspective. Recall Definition 4.3.3 that the program $\mathbb{P}$ determines a PLP model $[\![-]\!]_{\mathbb{P}} : \mathsf{Syn}_{\mathbb{P}} \to \mathbf{Stoch}(\mathbb{B})$, and the goal is to derive a BN model of the form $\mathsf{SynBN} \to \mathbf{Stoch}(\mathbb{B})$. The main insight is that, on the functorial level the aforementioned construction of Pr in Definition 4.3.22 is purely syntactical: it suffices to define a syntactic category $\mathsf{SynBN}_{G_{BN(\mathbb{P})}}$ together with a 'syntactic translation' functor $\mathcal{F} \colon \mathsf{SynBN}_{G_{BN(\mathbb{P})}} \to \mathsf{Syn}_{\mathbb{L}}$. Then the BN model is simply the composite functor $[\![-]\!]_{\mathbb{P}} \circ \mathcal{F} \colon \mathsf{SynBN}_{\mathbb{L}} \to \mathbf{Stoch}(\mathbb{B})$.

Definition 4.3.24. The category $\mathsf{SynBN}_{\mathbb{L}}$ is the freely generated CD category with the following generators:

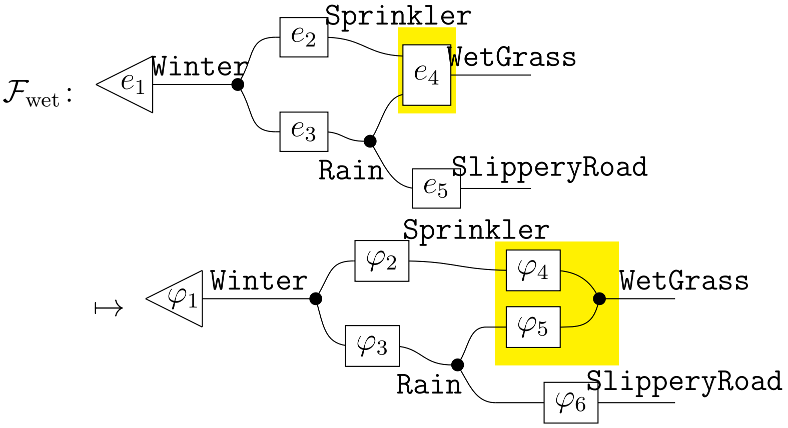
- the generating objects are At;
- the generating morphisms are the following set of string diagrams

$\Big\{$  $\;\Big|\; A \in At, \{B_1, \dots, B_m\} = \{B \mid \exists \varphi \in \mathbb{L} \text{ such that } A = \mathsf{head}(\varphi), B \in \mathsf{body}(\varphi)\}\Big\}$

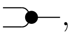
In words, there is one generating morphism e whose right port is A, and left ports contains exactly all the atoms B whose value may directly influence that of A via some clause. Note that this is exactly the definition of parental relations of $G_{BN(\mathbb{P})}$, and the syntax category $\mathsf{SynBN}_{\mathbb{P}}$ is exactly

Definition 4.3.25. The *translation functor* $\mathcal{F}\colon \mathsf{SynBN}_{\mathbb{L}} \to \mathsf{Syn}_{\mathbb{L}}$ is defined via the free construction of the domain category $\mathsf{SynBN}_{\mathbb{L}}$: $\mathcal{F}$ is identity on generating objects; on each generating morphism  , $\mathcal{F}(e) = comp_A$ (see Definition 4.3.8).

Example 4.3.26. Recall $\mathbb{P}_{\text{wet}}$ from Example 4.1.3. We illustrate the behaviour of $\mathcal{F}_{\text{wet}}\colon \mathsf{SynBN}_{\mathbb{L}_{\text{wet}}} \to \mathsf{Syn}_{\mathbb{L}_{\text{wet}}}$ via the following string diagrams, where the φ_is are as in Example 4.1.3:



In particular, the highlighted part in the bottom diagram is the $\mathcal{F}_{\text{wet}}$-image of the highlighted part from the top diagram, and it yields (via the interpretation) precisely the conditional probability distribution $\mathbb{B}_{\texttt{Sprinkler}} \times \mathbb{B}_{\texttt{Rain}} \to \mathcal{D}(\mathbb{B}_{\texttt{WetGrass}})$ in $\mathcal{B}_{\text{wet}}$ represented by the conditional probability distribution in (4.7).

Remark 4.3.27. Thanks to the separation of syntax and semantics, our construction is able to simplify and divide in two steps what in the literature (*cf.* [MSB08]) is performed in a single step. In the traditional approach, from a PLP $\mathbb{P}$, a DAG is constructed with 'AND' nodes and 'noisy-OR' nodes [MSB08], from which one derives the conditional probability distributions. Instead, we construct a simpler DAG $H = (V_H, \Sigma_H)$—in fact, a syntax category Syn_H^{BN} encoding H—and only as a next step we introduce a richer structure (modelled with , see Example 4.3.26) 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 $[\![-]\!]_{\mathbb{P}} : \mathsf{Syn}_{\mathbb{L}}^{PLP} \to \mathbf{Stoch}(\mathbb{B})$.

The correctness of the functorial construction is captured by the following statement of the equivalence of two BN models.

Proposition 4.3.28. $[\![-]\!]_{BN(\mathbb{P})} = [\![-]\!]_{\mathbb{P}} \circ \mathcal{F}$

Proof. We first check that these two functors have the same domain. In other words, we check that $\mathsf{SynBN}_{BN(\mathbb{P})} = \mathsf{SynBN}_{\mathbb{L}}$. Note that they are free CDMU categories with the same generating set of objects and morphisms: for every $A \in At$, there is precisely one generating morphism of the form $[B_1, \dots, B_m] \to [A]$, where $B_1, \dots, B_m$ are all those B_i such that there exists $\varphi \in \mathbb{L}$ satisfying $\mathsf{head}(\varphi) = A$ and $B \in \mathsf{body}(\varphi)$.

Then it suffices to show that the two functors behave the same on all generating objects and morphisms of their domain (syntactic) category. For generating objects, this is immediate by the definitions. So we focus on generating morphisms. Pick an arbitrary A with $pa(A) = \{B_1, \dots, B_k\}$ in $G_{BN(\mathbb{P})}$, and values $\mathbf{b} = (b_1, \dots, b_k) \in \mathbb{B}_{B_1} \times \cdots \times \mathbb{B}_{B_k}$, assume the corresponding generating morphism in $\mathsf{SynBN}_{G_{BN(\mathbb{P})}}$ is $e \colon [B_1, \dots, B_k] \to [A]$, we show that $[\![e]\!]_{BN(\mathbb{P})}(\mathbf{b}) = [\![\mathcal{F}(e)]\!]_{\mathbb{P}}(\mathbf{b})$, both as distributions over $\mathbb{B}_A$. Equivalently, the goal is to show that $[\![e]\!]_{BN(\mathbb{P})}(\mathbf{b})(1_A) = [\![\mathcal{F}(e)]\!]_{\mathbb{P}}(\mathbf{b})(1_A)$.

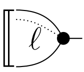
We assume that $\varphi_1, \dots, \varphi_\ell$ are all the $\mathbb{L}$-clauses such that $\mathsf{head}(\varphi) = A$, and $\psi_1, \dots, \psi_s$ are all the $\mathbb{P}$-clauses such that $\tau^{pr}(\psi)$ is one of those φ_i (note that $\ell \leq s$). Moreover, we suppose that $\psi_1, \dots, \psi_r$ are those ψ_j whose bodies are satisfied by the given $\mathbf{b}$.

Now, on one hand, to work directly with the Bayesian network $BN(\mathbb{P})$, by Definition 4.3.22 and Proposition 4.3.16, $[\![e]\!]_{BN(\mathbb{P})}(\mathbf{b})(1_A) = 1 - \left(\prod_{j=1}^{r}(1 - lab(\psi_j))\right)$.

On the other hand, using the syntactic transformation,

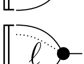
$$[\![\mathcal{F}(e)]\!]_{\mathbb{P}}(\mathbf{b}) = [\![comp_A]\!]_{\mathbb{P}}(\mathbf{b}) \qquad \text{(Def. 4.3.25)}$$

$$= \mu_{\mathcal{D}} \circ \mathcal{D}\Big([\![$$



$$]\!]_{\mathbb{P}}\Big)([\![\varphi_1]\!]_{\mathbb{P}}(\mathbf{b}) \otimes \cdots \otimes [\![\varphi_\ell]\!]_{\mathbb{P}}(\mathbf{b})) \qquad \text{(Def. 4.3.8)}$$

$$= \mu_{\mathcal{D}} \circ \mathcal{D}\Big([\![$$



$$]\!]_{\mathbb{P}}\Big)((1 - lab(\psi_1))|0_A\rangle + lab(\psi_1)|1_A\rangle, \dots,$$

$$(1 - lab(\psi_r))|0_A\rangle + lab(\psi_r)|1_A\rangle, 1|0_A\rangle, \dots, 1|0_A\rangle)$$

$$= \left(\prod_{j=1}^{r}(1 - lab(\psi_j))\right)|0_A\rangle + \left(1 - \left(\prod_{j=1}^{r}(1 - lab(\psi_j))\right)\right)|1_A\rangle \qquad \text{(Def. 4.3.3)}$$

This means that $[\![\mathcal{F}(e)]\!]_{\mathbb{P}}(\mathbf{b})(1_A) = 1 - \left(\prod_{j=1}^{r}(1 - lab(\psi_j))\right)$, so $[\![e]\!]_{BN(\mathbb{P})}(\mathbf{b}) = [\![\mathcal{F}(e)]\!]_{\mathbb{P}}(\mathbf{b})$. Therefore the two functors $[\![-]\!]_{BN(\mathbb{P})}$ and $[\![-]\!]_{\mathbb{P}} \circ \mathcal{F}$ are equal. □

The two constructions of this section are summarised by the following commutative diagrams (in the category Cat).

On one hand, the following diagram presents that we start with a PLP program $\mathbb{P}$ with underlying $\mathbb{L}$, transform it into a BN $BN(\mathbb{P})$, and then turn it into a PLP program $Prog(BN(\mathbb{P}))$

(with underlying inference structure $\mathbb{L}'$).

$$\begin{tikzcd}
\mathsf{Syn}_{\mathbb{L}} \arrow[d, "{[\![-]\!]_{\mathbb{P}}}"'] \arrow[r, phantom, "(1)", shift right=3ex] & \mathsf{SynBN}_{G_{BN(\mathbb{P})}} \arrow[l, "\mathcal{F}"'] \arrow[r, hook, "\iota"] \arrow[dl, "{[\![-]\!]_{BN(\mathbb{P})}}"'] \arrow[d, phantom, "(2)"] \arrow[u, phantom, "(3)"] & \mathsf{Syn}_{\mathbb{L}'} \arrow[ll, bend right, "\mathcal{G}"'] \arrow[dl, "{[\![-]\!]_{Prog(BN(\mathbb{P}))}}"] \\
\mathbf{Stoch}(\mathbb{B}) \arrow[r, equal] & \mathbf{Stoch}(\mathbb{B}) &
\end{tikzcd} \tag{4.8}$$

Given the definition of SynBN_G and $\mathsf{Syn}_{\mathbb{L}}$, we may let $\mathcal{G}$ be $\mathcal{F}$ plus the clause that $\mathcal{G}(d) = d$ for $d \in \{\sqsupset\!\!\bullet\!\!-, \bullet\!\!-\}$. Triangle (1) is the functorial definition of $[\![-]\!]_{BN(\mathbb{P})}$ (Proposition 4.3.28); square (2) states that $[\![-]\!]_{BN(\mathbb{P})}$ factors through its CDMU-lifting $[\![-]\!]_{Prog(BN(\mathbb{P}))}$ via the inclusion functor ι between the two syntactic categories; (3) connects the program $Prog(BN(\mathbb{P}))$ associated with $BN(\mathbb{P})$ with the original program $\mathbb{P}$.

On the other hand, the following diagram describes the situation when one starts with a Bayesian network $\mathcal{B}$, generates a PLP program $Prog(\mathcal{B})$ via canonical lifting (square (4)), and then construct the associated Bayesian network $BN(Prog(\mathcal{B}))$ (triangle (5)). The resulting BN is exactly $\mathcal{B}$ (diagram (6)).

$$\begin{tikzcd}
\mathsf{SynBN}_G \arrow[r, hook, "\iota"] \arrow[d, "{[\![-]\!]_{\mathcal{B}}}"] \arrow[rr, bend left, "="] \arrow[dr, phantom, "(4)"] & \mathsf{Syn}_{\mathbb{L}} \arrow[d, "{[\![-]\!]_{Prog(\mathcal{B})}}"] \arrow[u, phantom, "(6)"] \arrow[r, phantom, "(5)", shift right=3ex] & \mathsf{SynBN}_{G_{BN(Prog(\mathcal{B}))}} \arrow[l, "\mathcal{F}"'] \arrow[dl, bend left, "{[\![-]\!]_{BN(Prog(\mathcal{B}))}}"] \\
\mathbf{Stoch}(\mathbb{B}) \arrow[r, equal] & \mathbf{Stoch}(\mathbb{B}) &
\end{tikzcd} \tag{4.9}$$

4.4 Weighted logic programming

In this section we extend the above functorial approach to encompass weighted logic programming (WLP). We will provide a functorial semantics for WLP (Section 4.4.1), based on which one can express the least fixed-point semantics for WLP diagrammatically (Section 4.4.2). The similarity of the diagrams expressing certain semantic components in WLP with those in logic programs (Section 4.2.2) helps to formally establish the distinction between these two paradigms.

Let us fix a semiring $\mathbf{K} = \langle K, +, \cdot, \mathbf{0}, \mathbf{1}\rangle$ with a compatible lattice structure $\leq$ (Section 2.1.3). Also, we fix a weighted logic program $\mathbb{W}$ over $\mathbf{K}$, with atoms from a given set At. It might be helpful to take $\mathbf{K}$ as the min-sum semiring (Example 2.1.14) when reading through the following section, since the examples are all based on shortest-path problems over this semiring.

4.4.1 Functorial semantics of WLP

We start with the syntactic category. It is the same as that for LP and PLP, reflecting the intuition that the extension provided by WLP only affects the semantic dimension through the functorial perspective. We assume that the underlying inference structure of $\mathbb{W}$ is the definite program $\mathbb{L}$.

Definition 4.4.1. The syntactic category for WLP $\mathbb{W}$ is $\mathsf{Syn}_{\mathbb{L}}$, the freely generated CDMU category with generating objects and morphisms determined by $\mathbb{L} = \mathit{inf}(\mathbb{W})$.

Next we turn to the semantic category, which distinguishes WLP with LP and PLP. Intuitively the semantic category should reflect the fact that atoms are now interpreted no longer to Boolean values (as in LP and PLP), but instead take values in the semiring $\mathbf{K}$. Also, according to the least fixed-point semantics of WLP, clauses are read as monotone functions, thus the morphisms in this syntactic category should (at least) include monotone functions between the finite products of $\mathbf{K}$. Indeed, including only the monotone functions rather than all functions suffices for our current purpose. These justify the following choice of $\mathsf{MonFunc}(\mathbf{K})$ as the semantic cateogry.

Definition 4.4.2. The category $\mathsf{MonFunc}(\mathbf{K})$ has the following components:

- the objects are finite products of the form $\mathbf{K}_1 \times \cdots \times \mathbf{K}_k$, where each $\mathbf{K}_i$ is some copy of $\mathbf{K}$;
- the morphisms are monotone functions between the sets.

In particular, we denote the empty product as $\mathbf{1} = \{\bullet\}$. We need to make sure that this category $\mathsf{MonFunc}(\mathbf{K})$ has enough structure to interpret the CDMU syntactic category.

Proposition 4.4.3. $\mathsf{MonFunc}(\mathbf{K})$ *is a CDMU category.*

Proof. As a subcategory of Set, the symmetric monoidal structure of $\mathsf{MonFunc}(\mathbf{K})$ is inherited from that of Set, because the copier and discharger are both monotone functions. So we focus on the CDMU structure, which relies on the semiring structure of $\mathbf{K}$.

$$\begin{array}{rcccccrcccc}
\nabla & : & \mathbf{K} & \to & \mathbf{K}\times\mathbf{K} & \quad & \epsilon & : & \mathbf{K} & \to & \mathbf{1} \\
 & & x & \mapsto & (x,x) & & & & x & \mapsto & \bullet \\
\Delta & : & \mathbf{K}\times\mathbf{K} & \to & \mathbf{K} & & \eta & : & \mathbf{1} & \to & \mathbf{K} \\
 & & (x_1,x_2) & \mapsto & x_1+x_2 & & & & \bullet & \mapsto & \mathbf{0}
\end{array}$$

All these functions are monotone; in particular, Δ is monotone by the assumption that the ordering $\leq$ is compatible with the semiring structure, thus $x_1 \leq y_1$ and $x_2 \leq y_2$ implies $\Delta(x_1,x_2) = x_1 + x_2 \leq y_1 + y_2 = \Delta(y_1,y_2)$. To verify the CDMU axioms, we note that those for the comonoid structure (∇,ϵ) is the same as before. The correctness of the monoid structure axioms is below:

- Unital: $((id \otimes \eta) \mathbin{;} \Delta)(x) = \Delta(x, \mathbf{0}) = x + \mathbf{0} = x = id(x)$. Similarly $(\eta \otimes id) \mathbin{;} \Delta = id$.
- Associative: $((\Delta \otimes id) \mathbin{;} \Delta)(x, y, z) = x + y + z = ((id \otimes \Delta) \mathbin{;} \Delta)(x, y, z)$.
- Commutative: $(\gamma \mathbin{;} \Delta)(x, y) = \Delta(y, x) = x + y = \Delta(x, y)$.

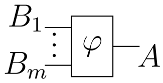
This completes the verification that $\mathsf{MonFunc}(\mathbf{K})$ is CDMU. □

Example 4.4.4. For the min-sum semiring **MinPlus** (see Example 2.2.7), its associated semantic category $\mathsf{MonFunc}(\mathbf{MinPlus})$ has comonoid structure:

$$\begin{array}{rcrcl} \Delta & : & (\mathbb{N} \cup \{+\infty\}) \times (\mathbb{N} \cup \{+\infty\}) & \to & \mathbb{N} \cup \{+\infty\} \\ & & (x_1, x_2) & \mapsto & \min(x_1, x_2) \end{array} \qquad \begin{array}{rcrcl} \eta & : & \mathbf{1} & \to & \mathbb{N} \cup \{+\infty\} \\ & & \bullet & \mapsto & +\infty \end{array}$$

Similar to the case of LP and PLP, a WLP program $\mathbb{W}$ (denote $\mathit{inf}(\mathbb{W})$ by $\mathbb{L}$) uniquely determines a functor $[\![-]\!]_{\mathbb{W}} : \mathsf{SynWLP}_{\mathbb{L}} \to \mathsf{MonFunc}(\mathbf{K})$, which maps each generating string diagram to (the semantics of) its corresponding WLP clause.

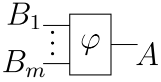
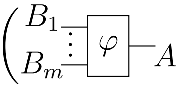
Definition 4.4.5. The functor $[\![-]\!]_{\mathbb{W}} : \mathsf{Syn}_{\mathbb{L}} \to \mathsf{MonFunc}(\mathbf{K})$ is defined as follows:

- On objects, $[\![A]\!]_{\mathbb{P}} \coloneqq \mathbf{K}_A$ for each $A \in At$.
- On morphisms, given each generating morphism  of $\mathsf{Syn}_{\mathbb{L}}$, $[\![\varphi]\!]_{\mathbb{W}}$ maps a state $(u_1, \ldots, u_m) \in \mathbf{K}_{B_1} \times \cdots \times \mathbf{K}_{B_m}$ to $lab(\psi) \cdot u_1 \cdot \cdots \cdot u_m$, where ψ is the (unique) $\mathbb{W}$-clause satisfying $\tau^{wt}(\psi) = \varphi$.
- $[\![-]\!]_{\mathbb{W}}$ maps the CDMU structure to the CDMU structure of $\mathsf{Set}(\mathbf{K})$ (see Proposition 4.4.3).

Following Proposition 4.4.3, $[\![-]\!]_{\mathbb{W}}$ is a CDMU functor. We call it a (WLP) model because of the following result relating WLP programs and such functors.

Proposition 4.4.6. *There is a 1-1 correspondence between weighted logic programs with inference structure* $\mathbb{L}$ *and generator-preserving CDMU functors* $\mathsf{Syn}_{\mathbb{L}} \to \mathsf{Set}(\mathbf{K})$.

Proof. We first define two constructions, from WLP to functors and the other way around, and then prove that they constitute the isomorphism.

The construction $[\![-]\!]_{\mathbb{W}}$ of an object-preserving CDMU functor from a WLP $\mathbb{W}$ is already defined in Definition 4.4.5. For the other direction, given an object-preserving CDMU functor $\mathcal{G}\colon \mathsf{SynWLP}_{\mathbb{L}} \to \mathsf{Set}(\mathbf{K})$, we define a WLP program $Prog(\mathcal{G})$ as follows. For each generating morphism , we define a clause $w :: A \leftarrow B_1, \ldots, B_k$. where $w = \mathcal{G}\Bigg($  $\Bigg)(\mathbf{1}_{B_1}, \ldots, \mathbf{1}_{B_k})$.

Next we show that $[\![-]\!]_{(\cdot)}$ and $Prog(\cdot)$ are inverse to each other.

- Starting from a WLP $\mathbb{W}$, and pick an arbitrary clause φ of the form $w :: A \to B_1, \dots, B_k$., $[\![\varphi]\!]_{\mathbb{W}}$ is a function $\mathbf{K}_{B_1} \times \cdots \times \mathbf{K}_{B_k} \to \mathbf{K}_A$ mapping $(b_1, \dots, b_k)$ to $w \cdot b_1 \cdot \cdots \cdot b_k$. Then the program $\mathit{Prog}([\![-]\!]_{\mathbb{W}})$ has exactly one clause ψ with $\tau^{wt}(\psi) = \varphi$, whose weight lable $\mathit{lab}(\psi) = [\![\varphi]\!]_{\mathbb{W}}\,(\mathbf{1}_{B_1}, \dots, \mathbf{1}_{B_k}) = w$. Thus we obtain the original clause that we start with. Moreover, every clause in $\mathit{Prog}([\![-]\!]_{\mathbb{W}})$ is generated in this way. So $\mathit{Prog}([\![-]\!]_{\mathbb{W}}) = \mathbb{W}$.

- Starting from an object-preserving CDMU functor $\mathcal{G}$, we get the program $\mathit{Prog}(\mathcal{G})$ such that for each generating morphism $\begin{smallmatrix}B_1 \\ \vdots \\ B_m\end{smallmatrix}\!\boxed{\varphi}\!-A$ in $\mathsf{SynWLP}_{\mathbb{L}}$, $\mathit{Prog}(\mathcal{G})$ has exactly one clause ψ with $\tau^{wt}(\psi) = \varphi$, and $\mathit{lab}(\psi) = \mathcal{G}\left(\begin{smallmatrix}B_1 \\ \vdots \\ B_m\end{smallmatrix}\!\boxed{\varphi}\!-A\right)(\mathbf{1}_{B_1}, \dots, \mathbf{1}_{B_k})$. Then the induced functor $[\![-]\!]_{\mathit{Prog}(\mathcal{G})}$ maps $\begin{smallmatrix}B_1 \\ \vdots \\ B_m\end{smallmatrix}\!\boxed{\varphi}\!-A$ to the function $\mathbf{K}_{B_1} \times \cdots \times \mathbf{K}_{B_k} \to \mathbf{K}_A$ mapping $(b_1, \dots, b_k)$ to $\mathit{lab}(\psi) \cdot b_1 \cdot \cdots \cdot b_k$, thus behaves the same on $\begin{smallmatrix}B_1 \\ \vdots \\ B_m\end{smallmatrix}\!\boxed{\varphi}\!-A$ as $\mathcal{G}$. So $[\![-]\!]_{\mathit{Prog}(\mathcal{G})} = \mathcal{G}$.

Therefore $[\![-]\!]_{(\cdot)}$ and $\mathit{Prog}(\cdot)$ witness the 1-1- correspondence between WLP programs with underlying inference structure $\mathbb{L}$ and generator-preserving CDMU functors $\mathsf{SynWLP}_{\mathbb{L}} \to \mathsf{Set}(\mathbf{K})$. □

Before going further with WLP, we find here a good place to postpone and discuss its connection with classical logic programming. Indeed it is well-known that propositional definite logic program is a special case of WLP, by taking the underlying semiring to be the Boolean semiring $\mathbf{B} = \langle \mathbb{B}, \vee, \wedge, 0, 1 \rangle$. Then a definite LP clause of the form $A \leftarrow B_1, \dots, B_k$. is semantically a function $\mathbb{B}_{B_1} \times \cdots \times \mathbb{B}_{B_k} \to \mathbb{B}_A$ mapping each $(b_1, \dots, b_k)$ to $1 \wedge b_1 \wedge \cdots \wedge b_k = b_1 \wedge \cdots \wedge b_k$, which is exactly the semantics of LP clauses. Under this view, Proposition 4.2.5 restricted to definite programs can be seen as a special case of Proposition 4.4.6, where $\mathbf{K} = \mathbf{B}$. However, general logic programs with negation cannot fit into the framework of WLP, at least not by taking the Boolean semiring $\mathbf{B}$. This can be seen by observing that the semantics of LP clauses with negations is not monotone.

Apart from that, an interesting observation regarding LP, PLP and WLP is that the semantic variation required for PLP and WLP are different. At the intuitive level, whereas the semantic category $\mathbf{Stoch}(\mathbb{B})$ for PLP can be thought as a 'Kleisli' variation of $\mathsf{Set}(\mathbb{B})$, in the semantic category $\mathsf{Set}(\mathbf{K})$ for WLP the morphisms of the 'basic' semantic category $\mathsf{Set}(\mathbb{B})$ remains as functions, but the objects change from $\mathbb{B}$ to $\mathbf{K}$. One potential implication of this observation is the possibility to combine these two variation to obtain a 'probabilistic weighted logic programming'. From a functorial perspective, this amounts to choosing a suitable semantic category, which is the '$\mathcal{D}$-Kleisli' variation of $\mathsf{MonFunc}(\mathbf{K})$. The application of such a formalism and its potential connection with Bayesian networks beyond Boolean-values remain to be studied.

4.4.2 Some semantic construct for WLP

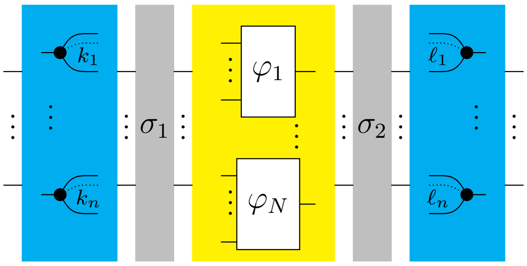
Analogously to what we did with LP and PLP, we conclude the current section on WLP by describing how certain semantics of WLP (Section 2.1.3) can be expressed with string diagrams of the syntax category. Our aim semantic constructs are the weighted version of immediate consequence operator and consequence operator. For simplicity, we dive into the freely generated traced CDMU category $\mathsf{SynWLP}^{tr}_{\mathbb{L}}$ defined as in Definition 4.2.10. Again, we need to check that our semantic category has enough structure to interpret the extra traces. Thanks to the choice of monotone functions, the semantic category for WLP is traced as well.

Proposition 4.4.7. $\mathsf{MonFunc}(\mathbf{K})$ *is a traced monoidal category.*

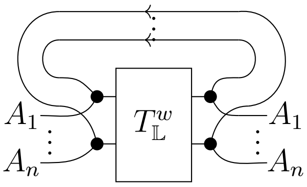
Proof. We define the trace as follows. For a morphism $f\colon X \times U \to Y \times U$, let $Tr^U_{X,Y}(f)$ be the function $X \to Y$ mapping $x \in X$ to $\pi_2 \circ f(x, u_x)$, where $u_x = \mu u.\pi_2 \circ f(x, u)$. The proof is similar to that for $\mathsf{MonFunc}(\mathbb{B})$ (Proposition 5.4.37), where essentially one replace $\mathbb{B}$ with the semiring $\mathbf{K}$ with a compatible ordering. □

The weighted immediate consequence operators — the weighted counterpart of consequence operators for logic programming — is represented by the following weighted immediate consequence diagram in the same favour as immediate consequence diagram $T_{\mathbb{L}}$.

Definition 4.4.8. The *weighted immediate consequence diagram* $T^w_{\mathbb{L}}$ is the following diagram (as a morphism of the form $[A_1, \ldots, A_n] \to [A_1, \ldots, A_n]$ in $\mathsf{Syn}^{tr}_{\mathbb{L}}$):

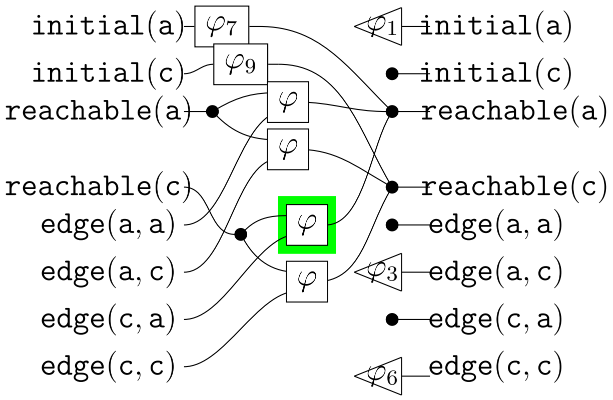


Definition 4.4.9. The *weighted consequence diagram* $C^w_{\mathbb{L}}$ is the following diagram (as a morphism of the form $[A_1, \ldots, A_n] \to [A_1, \ldots, A_n]$ in $\mathsf{Syn}^{tr}_{\mathbb{L}}$), where $T^w_{\mathbb{L}}$ is the weighted immediate consequence diagram from Definition 4.4.8.



Note that this diagram is exactly the same as the consequence diagram for logic programs (see Definition 4.2.15). This repeats the spirit of our functorial approach: the difference between some essential semantic constructs of various logic programming paradigms lies only in the semantic categories.

Example 4.4.10. We recall the program $\mathbb{P}_{sp}$ about shortest path problem from Example 2.1.16, based on the min-sum semiring **MinPlus** $= \langle \mathbb{N} \cup \{+\infty\}, \min, +, +\infty, 0\rangle$. We denote its underlying inference structure as $\mathbb{L}_{sp}$. Its weighted immediate consequence diagram $T^{w}_{\mathbb{L}_{sp}}$ — restricted to constants `a` and `c` — is as follows:



For instance, the green-highlighted diagram corresponds to the clause $0 ::$ `reachable(a)` $\leftarrow$ `reachable(c)`, `edge(c, a)`. in $\mathbb{P}_{sp}$.

Semantically, $\llbracket \bullet\!\!- \rrbracket_{\mathbb{W}} = \{(\bullet, +\infty)\}$, $\llbracket \sqsupset\!\!\bullet\!\!- \rrbracket_{\mathbb{W}} = \{((x_1, x_2), y) \mid \min(x_1, x_2) = y\}$; $\left\llbracket T^{w}_{\mathbb{P}_{sp} infStruct} \right\rrbracket_{\mathbb{W}}$ is a function such that, for a state u with $u(\texttt{reachable(c)}) = r$ and $u(\texttt{edge(c,a)}) = s$, $\left\llbracket T^{w}_{\mathbb{L}_{sp}} \right\rrbracket_{\mathbb{W}} (\texttt{reachable(a)}) = 0{+}r{+}s = r{+}s$, $\left\llbracket T^{w}_{\mathbb{L}_{sp}} \right\rrbracket_{\mathbb{W}} (\texttt{initial(a)}) = 0$, and $\left\llbracket T^{w}_{\mathbb{L}_{sp}} \right\rrbracket_{\mathbb{W}} (\texttt{initial(c)}) = +\infty$.

As one can expect, these diagrams express the desired operators for WLP programs.

Proposition 4.4.11. *The diagrams $T^{w}_{\mathbb{L}}$ and $C^{w}_{\mathbb{L}}$ express the weighted immediate consequence operator and the weighted consequence operator, respectively, in the following sense:*

$$\llbracket T^{w}_{\mathbb{L}} \rrbracket_{\mathbb{W}} = \mathbf{T}^{w}_{\mathbb{W}} \qquad\qquad \llbracket C^{w}_{\mathbb{L}} \rrbracket_{\mathbb{W}} = \mathbf{C}^{w}_{\mathbb{W}}$$

Proof. The proof is similar to that for LP in Subsection 4.2. For weighted immediate consequence operator, for each $(a_1, \ldots, a_n) \in \mathbf{K}_{A_1} \times \cdots \times \mathbf{K}_{A_k}$, the j-th component of $\llbracket T^{w}_{\mathbb{L}} \rrbracket_{\mathbb{W}} (a_1, \ldots, a_n)$ is

$$\sum_{\psi = A \leftarrow A_{i_1}, \ldots, A_{i_k}. \in \mathbb{W} \text{ s.t. } A = A_j} lab(\psi) \cdot a_{i_1} \cdot \cdots \cdot a_{i_k}$$

thus $\llbracket T^{w}_{\mathbb{L}} \rrbracket_{\mathbb{W}} = \mathbf{T}^{w}_{\mathbb{W}}$.

For the weighted consequence operator,

$$\begin{aligned} \llbracket C^{w}_{\mathbb{L}} \rrbracket_{\mathbb{W}} (a_1, \ldots, a_n) &= \left\llbracket Tr^{[A_1,\ldots,A_n]}_{[A_1,\ldots,A_n],[A_1,\ldots,A_n]} \left(\sqsupset\!\!\bullet\!\!-_{[A_1,\ldots,A_n]} \,;\, T^{w}_{\mathbb{L}} \,;\, -\!\!\bullet\!\!\sqsubset_{[A_1,\ldots,A_n]} \right) \right\rrbracket_{\mathbb{W}} (a_1, \ldots, a_n) \\ &= \llbracket \pi_1 \circ (\sqsupset\!\!\bullet\!\!- \,;\, T^{w}_{\mathbb{L}} \,;\, -\!\!\bullet\!\!\sqsubset)(\mathbf{a}, \mathbf{u_a}) \rrbracket_{\mathbb{W}} \end{aligned}$$

where $\mathbf{u_a} = \mu \mathbf{u}.\pi_2 \circ \llbracket \sqsupset\!\!\bullet\!\!- \,;\, T^{w}_{\mathbb{L}} \,;\, -\!\!\bullet\!\!\sqsubset \rrbracket_{\mathbb{W}} (\mathbf{a}, \mathbf{u})$. Thus

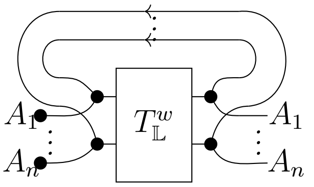
$$\mathbf{u_a} = \mu \mathbf{u}.\pi_2 \circ \llbracket -\!\!\bullet\!\!\sqsubset \rrbracket_{\mathbb{W}} \circ \llbracket T^{w}_{\mathbb{L}} \rrbracket_{\mathbb{W}} \circ \llbracket \sqsupset\!\!\bullet\!\!- \rrbracket_{\mathbb{W}} (\mathbf{a}, \mathbf{u})$$

$$
\begin{aligned}
&= \mu\mathbf{u}.\, \llbracket T^w_{\mathbb{L}} \rrbracket_{\mathbb{W}} \circ \llbracket \supset\!\!\bullet\!\!- \rrbracket_{\mathbb{W}} (\mathbf{a}, \mathbf{u}) \\
&= \mu\mathbf{u}.\, \llbracket T^w_{\mathbb{L}} \rrbracket_{\mathbb{W}} (\mathbf{a} + \mathbf{u}) = \mu\mathbf{u}.\mathbf{T}^w_{\mathbb{W}}(\mathbf{a} + \mathbf{u}) \\
&= \mathbf{C}^w_{\mathbb{W}}(\mathbf{a})
\end{aligned}
$$

This verifies that $\llbracket C^w_{\mathbb{L}} \rrbracket_{\mathbb{W}} = \mathbf{C}^w_{\mathbb{W}}$. □

As a special case, the canonical least model semantics of WLP programs can be recovered by sending the trivial input to the weighted consequence operator.

Corollary 4.4.12. *The following diagram expresses the least model semantics of WLP:*



(4.10)

Proof. Since $\llbracket C^w_{\mathbb{L}} \rrbracket_{\mathbb{W}} = \mathbf{C}^w_{\mathbb{W}}$ (Proposition 4.4.11), and $\llbracket \bullet\!\!- \rrbracket_{\mathbb{W}} = \{(\bullet, \mathbf{0})\}$, the statement follows immediately from the fact that the canonical semantics is exactly $\mathbf{C}^w_{\mathbb{W}}(\mathbf{0}, \ldots, \mathbf{0})$. □

Example 4.4.13. Coming back to Example 4.4.10, one can calculate that the diagram (4.10) for $\mathbb{P}_{sp}$ is interpreted as the function mapping $\bullet \in \mathbf{1}$ to the state where, for instance, `initial(a)` $=$ 0, `initial(c)` $= +\infty$, `reachable(a)` $=$ `0`, `reachable(b)` $= 9$. Indeed, the values for `reachable(a)`, `reachable(b)` and `reachable(c)` are exactly the shortest paths from the starting node `a` to node `a`, `b`, `c`, respectively.

Comparing the current part with the diagrammatic presentation of semantics for LP (Section 4.2.2) and PLP (Section 4.3.2), we notice that WLP is closer to the former, and more specifically, to the special case of definite logic programming. This is clear from the similarity of the (immediate) consequence operators, the selection of the appropriate semantic category to interpret the traced diagrammatic language, and the statement about certain diagrams expressing the canonical semantics. It is mainly because both semantics have a straightforward definition as the least point of certain monotone functions over a suitable semantic domain. While the semantics for logic programs with PLP and PLP is not simply defined like that, here are some potential directions for extending the current approach to involve these two cases.

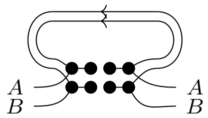
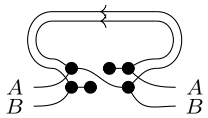
For logic programs with negation, while the immediate consequence operator is *not* monotone on the standard Boolean domain $\mathbb{B}$, there is a benchmark semantics based on least-fixed point construction called well-founded semantics [VGRS91b, Prz90]. Essentially one needs a three-value domain instead of $\mathbb{B}$, with an additional truth value called `unknown`, representing that the truth value is not decided. In this setting, there is a monotone operator playing the role of immediate consequence operator, and the well-founded semantics is the least fixed point which intuitively only assigns truth values 1 and 0 to atoms that can be proved or disproved, and leave other atoms as `unknown`. Though we are not going into any detail, we point out that one subtlety is that two different ordering structure is implicitly used in the well-founded semantics construction.

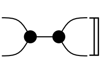
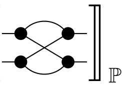
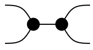
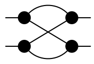
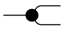
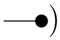
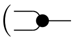
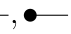
For probabilistic logic programs, the canonical distribution semantics is not constructed using certain fixed-point construction. It is something simpler: one randomly chooses the clauses only once, which is different from the semantics of probabilistic while-loops, for instance in Kozen [Koz83]. One potential extension is to invent a novel semantics in which the probabilities of clauses are counted at every single usage, possibly via certain least-fixed point construction. While the category of stochastic matrices **Stoch**($\mathbb{B}$) is not traced, one might extend to the traced category of stochastic matrices of non-negative real numbers, of which **Stoch**($\mathbb{B}$) is a subcategory. We leave these potential interesting development to future work.

Chapter 5

An algebra for propositional logic programs

In this section we present a sound and complete diagrammatic calculus to reason about equality between propositional definite logic program. Recall the functorial semantics for (probabilistic, weighted) logic programs from Chapter 4. While the separation of the syntactic categories of string diagrams from the semantics helps to clarify the roles of syntactic construction and semantics operations, one restriction is that one can only *present* certain constructs in diagrammatic form but not *reason about* them. For example, consider the two logic programs $\mathbb{P} = \{A \leftarrow ., B \leftarrow .\}$ and $\mathbb{Q} = \{A \leftarrow ., B \leftarrow A.\}$. While $\mathbb{P}$ and $\mathbb{Q}$ have the same Herbrand semantics, yet this equivalence is not derivable simply from the diagrammatic representation of their consequence diagrams (Definition 4.2.15):

$C_{\mathbb{P}} =$  $C_{\mathbb{Q}} =$ 

In fact the CDMU axioms for the string diagrams only capture very minimal properties of their interpretations. For instance, under the interpretation $[\![-]\!]_{\mathbb{P}} : \mathsf{Syn}^{PLP}_{\mathbb{L}} \to \mathsf{Set}(\mathbb{B})$ from Definition 4.2.3, the semantic equation $[\![$  $]\!]_{\mathbb{P}} = \left[\!\!\left[\right.\right.$  $\left.\left.\right]\!\!\right]_{\mathbb{P}}$ holds, since both are equal to the function $\lambda(x_1, x_2).(x_1 \vee x_2, x_1 \vee x_2) \colon \mathbb{B} \times \mathbb{B} \to \mathbb{B} \times \mathbb{B}$. Yet  $=$  is not derivable in the syntactic category: in fact the CDMU axioms do not claim any relation between the comonoid structure ( , ) and monoid structure ( , ).

The goal is then to equip the diagrams with an algebraic theory, such that properties such as the equivalence of programs can be solved using solely diagrammatic reasoning. In this section we achieve the goal for propositional definite logic programs. In Section 5.1 we introduce the target semantic objects and the diagrammatic calculus *SATA* (**Sat**isfiability **a**lgebra). Next we explore some properties of *SATA* (Section 5.2). We take a necessary detour to discuss matrix diagrams in Section 5.3 before proving the completeness result in Section 5.4 and Section 5.5.

Finally, we return to logic programs and exemplify how *SATA* enables to prove equivalence of programs diagrammatically. One final remark on the notation. In the rest of this chapter we will use $a, b, \dots$ instead of capital letters to denote atoms.

5.1 *SATA*

In this subsection we introduce a diagrammatic calculus *SATA* to reason about monotone relations over finite Boolean algebras (FBA) (Section 5.1.1). The name *SATA* refers to its invention as a **sat**isfiability **a**lgebra, proposed in Gu et al. [GPZ22] for picturing Boolean satisfiability. Given the standard semantics of propositional definite logic programs as certain least fixed points over the space of Boolean valuations, *SATA* can be used to encode such semantics, and convert program equivalence into diagrammatic equality.

Our diagrammatic calculus is mainly inspired by the line of research on graphical linear algebra (GLA) (Section 2.3.1). In particular, we adopt the same idea as in Piedeleu and Zanasi [PZ21] to work in some categories of *monotone relations* to characterise least fixed point semantics. So before diving into the calculus, we introduce the semantics and the corresponding categories.

5.1.1 Monotone functions and monotone relations

As mentioned above, the semantic domain of interest is the category of *monotone relations* between finite Boolean algebras. We first recall the notion of monotone relations, which can be defined more generally on arbitrary preorders.

Definition 5.1.1. A *monotone relation* R between two preordered sets $\langle X, \leq_X \rangle$ and $\langle Y, \leq_Y \rangle$ is a relation $R \subseteq X \times Y$ that satisfies the following monotonicity condition: if $(x, y) \in R$, then for arbitrary $x' \in X$ and $y' \in Y$, $x' \leq_X x$ and $y \leq_Y y'$ implies $(x', y') \in R$.

We would like to mention one useful intuition for monotone relations in terms of resource, from [FS19]: by viewing X and Y as products and resources respectively, $(x, y) \in R$ means that y-many Y-resource is enough to produce x-many X-product, thus any Y-resource y' larger than y (namely $y' \geq y$) can produce any x' many X-product less than x (namely $x' \leq x$), thus $(x', y') \in R$.

Example 5.1.2. For any preordered sets $\langle X, \leq_X \rangle$ and $\langle Y, \leq_Y \rangle$, the total relation $R = X \times Y$ and the empty relation $\varnothing$ are always monotone.

Consider the set of natural numbers $\mathbb{N}$ with the standard ordering $\leq$. The relation $R = \{(x, y) \in \mathbb{N} \times \mathbb{N} \mid x \leq 1, y \geq 99\}$ is a monotone relation, while $S = \{(x, y) \in \mathbb{N} \times \mathbb{N} \mid x \leq 1, y \text{ is prime}\}$ is not monotone.

From a categorical point of view, preorders and monotone relations form a category: composition is relation composition, and the identity morphism $id \colon \langle X, \leq_X \rangle \to \langle X, \leq_X \rangle$ is simply the relation $\leq_X$ itself. We denote this category as MonRel, the category of **mon**otone **rel**ations. Moreover, similar to that $\mathsf{Rel}_\times$ with products and singleton sets form a SMC, MonRel has a similar symmetric monoidal structure.

Definition 5.1.3 ([FS19])**.** MonRel together with the following structure form a symmetric monoidal category:

- The tensor product $\otimes$ on objects is products of preordered sets; $\otimes$ on morphisms is product of relations;
- The tensor unit is the singleton set $\{*\}$ together with the trivial (and only) preorder on it $\{(*,*)\}$;
- The swapping natural transformation σ on $\langle X, \leq_X\rangle$ and $\langle Y, \leq_Y\rangle$ is the monotone relation $\sigma_{X,Y} = \{((x_1, y_1), (y_2, x_2)) \mid x_1 \leq x_2, y_1 \leq y_2, x_1, x_2 \in X, y_1, y_2 \in Y\}$.

MonRel is exactly the category of Boolean-profunctors in Fong and Spivak [FS19], a simplification of the (2-)category of profunctors where Boolean-values replace set-values. Moreover, MonRel is a 2-category, where morphisms are poset-enriched by the inclusion relation $\subseteq$: given two MonRel-morphisms $R, S \colon X \nrightarrow Y$ $R \subseteq S$ if whenever $(x, y) \in R$, $(x, y) \in S$ as well.

To our purpose, monotone relations are important because they can be used to represent monotone functions, in a similar fashion that one can use relations to encode functions as their graphs.

Definition 5.1.4. Let $f \colon X \to Y$ be a monotone function between preordered sets $\langle X, \leq_X\rangle$ and $\langle Y, \leq_Y\rangle$. A monotone relation $R \colon X \nrightarrow Y$ *represents* f if $(x, y) \in R$ precisely when $f(x) \leq y$.

It is straightforward to verify that, if a relation R represents some monotone function f, then R is monotone: suppose $(x, y) \in R$, then by definition $f(x) \leq y$, so for arbitrary x', y' satisfying $x' \leq x$ and $y \leq y'$, $f(x') \leq f(x) \leq y \leq y'$, which implies that $(x', y') \in R$ as well.

Example 5.1.5. Continuing 5.1.2, if preordered set $\langle Y, \leq_Y\rangle$ has a least element $\perp_Y$, then the total relation as a monotone relation represents the constant function $x \in X \mapsto \perp_Y$.

In category theory language, we formalise this notion of 'representation' as a symmetric monoidal functor from the category of monotone functions MonFun to MonRel.

Definition 5.1.6. The category of monotone function MonFun has preordered sets as objects, and monotone functions as morphisms. Identity morphisms are identity functions, and morphism composition is function composition.

MonFun together with products of preorders and functions form a SMC. Then the representation of monotone functions as monotone relations can be defined formally as a symmetric monoidal functor $\mathsf{MonFun} \to \mathsf{MonRel}$.

Proposition 5.1.7. *The functor* $\mathcal{R}\colon \mathsf{MonFun} \to \mathsf{MonRel}$ *defined below is a symmetric monoidal functor:*

- *On objects,* $\mathcal{R}(\langle X, \leq_X\rangle) = \langle X, \leq_X\rangle$*, namely* $\mathcal{R}$ *is identity on objects.*
- *On morphisms, given a monotone function* $f\colon \langle X, \leq_X\rangle \to \langle Y, \leq_Y\rangle$*,* $\mathcal{R}f$ *is the monotone relation generated by* f*:* $(x, y) \in \mathcal{R}(f)$ *iff* $f(x) \leq y$*.*

Proof. The proof consists of two parts. First, $\mathcal{R}$ is a functor. Second, $\mathcal{R}$ respects the SMC structure.

For the functoriality of $\mathcal{R}$, it suffices to verify that if R, S represent monotone functions $f\colon X \to Y$ and $g\colon Y \to Z$ respectively, then $R \fatsemi S$ represents $g \circ f$.

For the second part, it suffices to show that $\mathcal{R}(f \times g) = \mathcal{R}(f) \times \mathcal{R}(g)$, and $\mathcal{R}(\sigma) = \sigma'$ (where *sym* and σ' are the symmetric structure of MonFun and MonRel, respectively). For example, to see that σ' represents σ, $\sigma(x, y) = (y, x)$ $((x, y), (y', x')) \in \sigma'$ iff $x \leq x'$ and $y \leq y'$ iff $\sigma(x, y) = (y, x) \leq (y', x')$. □

Unlike the case of functions and their representation as relations, in general such representation of monotone functions as monotone relations is not faithful: two distinct functions may have the same relational representation. But when ordering is a partial order, such representation is faithful.

Example 5.1.8. To illustrate that such representation is *not* necessarily faithful for preordered sets in general, consider the preordered (and *not* partially ordered) set depicted $\langle A, \leq\rangle$ as below, where an arrow $x \to y$ means $x \leq y$ (with reflexive arrows omitted).

$$\begin{tikzcd} a \arrow[r, bend left] & b \arrow[l, bend left] \end{tikzcd}$$

Let f and g be two functions $A \to A$ such that f is identity on A, and g maps both a and b to a. Then f and g are both monotone, yet they are represented by the same monotone relation $A \times A$, namely the total relation.

Proposition 5.1.9. *Suppose* $\langle X, \leq_X\rangle$ *and* $\langle Y, \leq_Y\rangle$ *are partially ordered sets, and* f, g *are both monotone functions* $X \to Y$*. If monotone relation* $R\colon X \nrightarrow Y$ *represents both* f *and* g*, then* $f = g$*.*

Proof. Assume that for some monotone functions $f, g\colon X \to Y$, both of them are represented by a monotone relation R, we show $f = g$. For an arbitrary $x \in X$, $(x, f(x))$ and $(x, g(x))$ are both in R. $(x, g(x)) \in R$ and R represents f imply that $f(x) \leq g(x)$. Similarly, $(x, f(x)) \in R$ and R represents g imply that $g(x) \leq f(x)$. Since $\leq_Y$ is a partial order, this entails that $f(x) = g(x)$. Then $f(x) = g(x)$ for arbitrary $x \in X$ means that $f = g$. □

In categorical terms, Proposition 5.1.9 amounts to the following statement about functor $\mathcal{R}$.

Proposition 5.1.10. *Let* $\mathsf{MonFun}_{\mathsf{po}}$ *(resp.* $\mathsf{MonRel}_{\mathsf{po}}$*) be the full subcategory of* MonFun *(resp.* MonRel*) where objects are partially ordered sets. Then* $\mathcal{R}\colon \mathsf{MonFun}_{\mathsf{po}} \to \mathsf{MonRel}_{\mathsf{po}}$ *is faithful.*

The representation also preserves the ordering structure on monotone functions and monotone relations, respectively. That is, $\mathcal{R}$ respects the 2-category structure on $\mathsf{MonFun}_{\mathsf{po}}$ and MonRel_{po} (contravariantly).

Proposition 5.1.11. *For arbitrary* $\mathsf{MonFun}_{\mathsf{po}}$*-morphisms* $f, g\colon X \to Y$*, if* $f \leq g$*, then* $\mathcal{R}(f) \supseteq \mathcal{R}(g)$*.*

Proof. Given two monotone functions $f, g\colon X \to Y$ satisfying $f \leq g$, $\mathcal{R}(f)$ and $\mathcal{R}(g)$ are their representing monotone relations, respectively. For any $(x, y) \in \mathcal{R}(g)$, by definition, $g(x) \leq y$, so $f(x) \leq g(x) \leq y$, which entails $(x, y) \in \mathcal{R}(f)$ as well. □

Finally we are able to specify the semantics category that we are interested in. We use $\mathbb{B}$ to denote the two-element Boolean algebra consisting of $0, 1$ and $0 \leq 1$. It is a basic result in universal algebra that $\mathbb{B}$ is the only nontrivial product irreducible finite Boolean algebra.

Theorem 5.1.12 ([Hun04])**.** *Every finite Boolean algebra* $\mathbb{A}$ *is a finite product of* $\mathbb{B}$*: there exists* $k \in \mathbb{N}$ *such that* $\mathbb{A} \cong \mathbb{B}^k$*.*

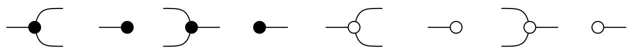
Definition 5.1.13. The *category of monotone relations over finite Boolean algebras* $\mathsf{MonRel}_{\mathbb{B}}$ is the full subcategory of MonRel_{po} whose objects are finite products of $\mathbb{B}$, say $\mathbb{B}^n$. Similarly, the *category of monotone functions over finite Boolean algebras* $\mathsf{MonFun}_{\mathbb{B}}$ is the full subcategory of $\mathsf{MonFun}_{\mathsf{po}}$ whose objects are finite products of $\mathbb{B}$.

5.1.2 Inequational theory *SATA*

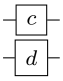
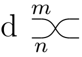
In this subsection we present the diagrammatic calculus *SATA* that is sound and completeness wrt the inclusion relation $\subseteq$ in $\mathsf{MonRel}_{\mathbb{B}}$. The syntax is a prop, freely generated by a signature containing black and white generators.

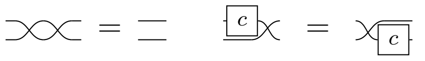
Definition 5.1.14. The category Syn is the prop freely generated by the following ingredients:

- Objects: a single generating object 1;
- Morphisms: the following generating morphisms depicted as string diagrams, where we adopt the convention to read from left to right, top to bottom:



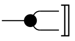
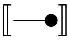
(5.1)

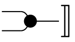
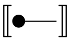
The symmetric monoidal structure on Syn consists of the monoidal product $\oplus$, unit I, and swapping σ, which we spell out now. We follow the convention in [Zan15] to denote the monoidal product of $1^{\oplus n}$ of n copies of the generating object 1 simply by n. The monoidal product of two string diagrams is simply their juxtaposition, say $c \oplus d$ is . The identity morphism $id_n \colon n \to n$ and swapping $\sigma_{m,n} \colon m \oplus n \to n \oplus m$ are depicted as string diagrams and , respectively, and we omit 1 on the (co)domains. As a freely generated (strict) SMC, Syn only satisfies the axioms of symmetric monoidal categories, depicted as equations between string diagrams. In particular, the symmetric structure needs to satisfy the following conditions, where c ranges over all Syn-morphisms:

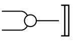
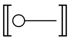
 (5.2)

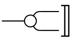
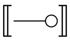
We interpret Syn as FBAs and monotone relations between them, and following the functorial semantics approach we define it formally as a functor between the syntax category Syn and the semantics category MonRel.

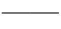
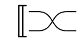
Definition 5.1.15. We define the functor $[\![\cdot]\!] : \mathsf{Syn} \to \mathsf{MonRel}$ as the (lax) symmetric monoidal category freely generated obtained by mapping the generating objects 1 to $\mathbb{B}$, and the generating morphisms as follows:

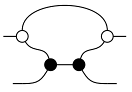
$[\![$$]\!] = \{(x,(y_1,y_2)) \mid x \le y_i, \text{ and } x, y_1, y_2 \in \mathbb{B}\}$ $\quad$ $[\![$$]\!] = \{(x, \bullet) \mid x \in \mathbb{B}\}$

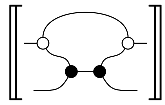
$[\![$$]\!] = \{((x_1,x_2),y) \mid x_i \le y, \text{ and } x_1, x_2, y \in \mathbb{B}\}$ $\quad$ $[\![$$]\!] = \{(\bullet, x) \mid x \in \mathbb{B}\}$

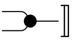
$[\![$$]\!] = \{((x_1,x_2),y) \mid x_1 \wedge x_2 \le y, \text{ and } x_1, x_2, y \in \mathbb{B}\}$ $\quad$ $[\![$$]\!] = \{(\bullet, 1)\}$

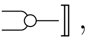
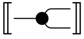
$[\![$$]\!] = \{(x,(y_1,y_2)) \mid x \le y_1 \vee y_2, \text{ and } x, y_1, y_2 \in \mathbb{B}\}$ $\quad$ $[\![$$]\!] = \{(0, \bullet)\}$

It is helpful to spell out more details of the functor $[\![\cdot]\!]$ in Definition 5.1.15. By 'freely generated', functor $[\![\cdot]\!]$ is well-defined using the free construction of the domain category Syn: for each natural number n, $[\![n]\!] = \mathbb{B}^n$; $[\![$$]\!] = \{(x,y) \mid x \le y \text{ and } x, y \in \mathbb{B}\}$; $[\![$$]\!] = \{((x_1,x_2),(y_1,y_2)) \mid x_1 \le y_2, x_2 \le y_1\}$; if $[\![c]\!]$ and $[\![d]\!]$ are already defined for some Syn-morphisms c and d, then $[\![c \,;\, d]\!] \coloneqq [\![c]\!] \fatsemi [\![d]\!]$ (whenever composable) and $[\![c \otimes d]\!] \coloneqq [\![c]\!] \times [\![d]\!]$, where $\fatsemi$ and $\times$ are the composition and products of relations, respectively.

Example 5.1.16. Take the diagram  as an example. Its interpretation can be compositionally computed as follows:

$((x_1,x_2),(y_1,y_2)) \in$ 

$\iff \exists u_1,u_2,u_3,u_4 \in \mathbb{B} : (x_1,(u_1,u_2)) \in [\![$$]\!]$, $((u_2,x_2),u_3) \in [\![$$]\!]$,

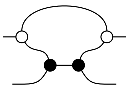
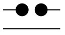
$((u_1,u_4),y_1) \in [\![$$]\!]$, $(u_3,(u_4,y_2)) \in [\![$$]\!]$

$\iff \exists u_1,u_2,u_3,u_4 \in \mathbb{B} : x_1 \le u_1 \vee u_2, u_2 \vee x_2 \le u_3, u_1 \wedge u_4 \le y_1, u_3 \le u_4 \wedge y_2$

$\iff \exists u_1,u_2,u_4 : x_1 \le u_1 \vee u_2, u_2 \vee x_2 \le u_4 \wedge y_2, u_1 \wedge u_4 \le y_1$

$\iff x_2 \le y_2$

In general, for the interpretation $[\![\cdot]\!]$ with a freely generated domain category, there is nothing preventing two different Syn-morphisms to have the same semantics.

Example 5.1.17. Recall the diagram  from Definition 5.1.16. Its interpretation is the same as that of the simpler diagram  .

The goal is then to provide an algebraic theory such that two diagrams have the same interpretation under $[\![\cdot]\!]$ precisely when they are equal in the theory. In other words, we aim at a sound and complete theory wrt $[\![\cdot]\!]$. In our case, the semantics is essentially *inequational* because diagrams are interpreted as relations and compared via inclusion between relations, so we give an inequational theory *SATA* in Figure 5.2.

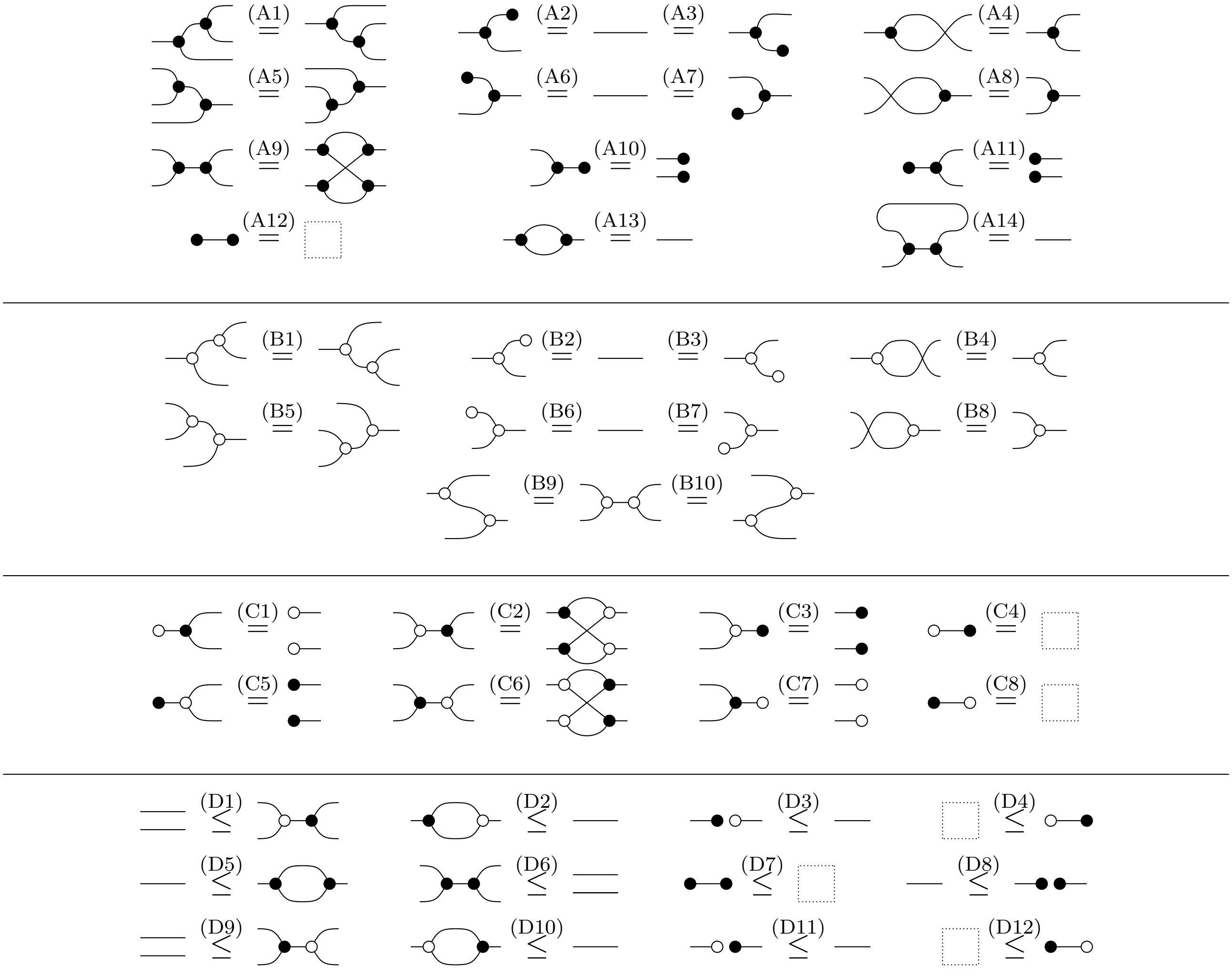


Figure 5.1: The equational theory *SATA*

Theorem 5.1.18. *For any* Syn*-morphisms* $c, d\colon m \to n$*,* $[\![c]\!] \subseteq [\![d]\!]$ *if and only if* $c \leq_{SATA} d$*.*

In the remainder of this section, we shall often omit the subscription *SATA* in $\leq_{SATA}$ (*resp.* $=_{SATA}$) and simply write $c \leq d$ (*resp.* $c = d$) when no confusion arises. Also, we take $\leq_{SATA}$ to be primary, and regard $c =_{SATA} d$ as abbreviation of $c \leq_{SATA} d$ and $d \leq_{SATA} c$, or equivalently, $\leq_{SATA}$ is a partial ordering. Using category theory language, such soundness and completeness result as Theorem 5.1.18 can be neatly stated as an isomorphism between a syntactic category and a semantic category: while the semantic category is $\mathsf{MonRel}_{\mathbb{B}}$, the syntactic category is Syn quotiented by the inequational theory $SATA$, thus also a poset-enriched category.

Definition 5.1.19. $\mathbb{SATA}$ is the poset-enriched (2-)category whose components are:

- 0-objects: objects of Syn;
- 1-objects: morphisms of Syn quotiented by equations derivable from $SATA$;
- 2-objects: given two 1-objects $c, d\colon m \to n$, there is a 2-object $c \to d$ if and only if $c \leq d$ is derivable from $SATA$.

In short, it is a poset-enriched category quotiented by some inequational theory. Theorem 5.1.18 can then be restated as the following isomorphism of 2-categories, whose proof we leave to Section 5.5.

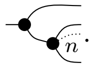
Theorem 5.1.20. $\mathbb{SATA} \cong \mathsf{MonRel}_{\mathbb{B}}$

Before moving forward to the technical development, let us provide some intuition and explanation of the $SATA$-axioms. While they share many properties with GLA, $SATA$ has some crucial differences due to its inequational nature.

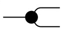
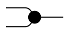
Block A axioms. (A1)-(A4) state that —●⊏ and —● form a commutative comonoid: (A1) says —●⊏ is associative so, in algebraic terms, brackets are unnecessary; (A2)-(A3) say —● is the unit for —●⊏; (A4) says —●⊏ is commutativity so the order of wires on the right port of —●⊏ does not matter. Similarly, (A5)-(A8) state that (⊐●—, ●—) forms a commutative monoid.

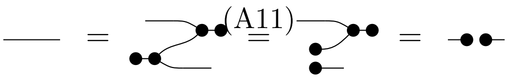
The associativity axiom (A1) for —●⊏ justifies introducing the following generalised n-ary operations —●⊏$_n$ as syntactic sugar.

Definition 5.1.21. The Syn-morphism —●⊏$_n$: $n \to 0$ is defined inductively on natural number n as follows:

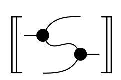
- —●⊏$_0$ = —●; —●⊏$_1$ = ——; —●⊏$_2$ == —●⊏;
- For $n \geq 2$, if —●⊏$_n$ is defined, then —●⊏$_{n+1}$ = .

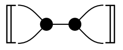
A similar definition works for the syntactic sugar $_n$⊐●— standing for 'n cocopies', and later for —○⊏$_n$ and $_n$⊐○— . (A9)-(A12) state that (—●⊏, —●, ⊐●—, ●—) forms a bimonoid (also called a bialgebra) [Lac04]. (A13) says this bimonoid *degenerates* or *is idempotent.* (A14) uses a syntax sugar that will be soon introduced in Definition 5.1.22. Basically it says that a 'void' feedback does not has any effect, thus simply an identity morphism.

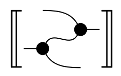
We would also like to point out that the ●-structure does *not* form a Frobenius structure, as is common for copy and cocopy structure in GLA and its close relatives. Technically, that (, ) forms a bimonoid means that it cannot be Frobenius, because being bimonoid and Frobenius at the same time entails that the theory would be inconsistent, in the sense that all morphisms of the same type are equal:

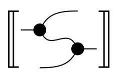
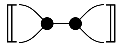
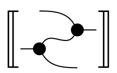
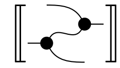
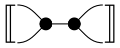
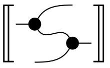


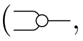
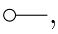
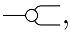
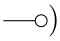
Semantically, we can work out the semantics of the diagrams in an Frobenius equation like (B9) and (B10), and convince ourselves of their difference:

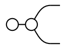
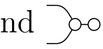

$$= \{((x_1, x_2), (y_1, y_2)) \mid \exists u : x_1 \leq y_1 \wedge u, u \vee x_2 \leq y_2\}$$


$$= \{((x_1, x_2), (y_1, y_2)) \mid \exists v : x_1 \vee x_2 \leq v, v \leq y_1 \wedge y_2\}$$
$$= \{((x_1, x_2), (y_1, y_2)) \mid x_1 \vee x_2 \leq y_1 \wedge y_2\}$$


$$= \{((x_1, x_2), (y_1, y_2)) \mid \exists w : x_1 \vee v \leq y_1, x_2 \leq v \wedge y_2\}$$

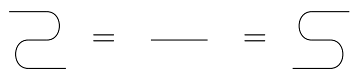
For instance, $((0, 1), (0, 1))$ is in  but not in  nor ; symmetrically, $((1, 0), (1, 0))$ is in  but not in  nor .

Block B axioms. (B1)-(B10) state that (, , , ) forms a commutative Frobenius algebra [CW87]. A first important consequence is that this structure makes 𝕊𝔸𝕋𝔸 a *compact closed category* [KL80], a category in which one can interpret fixed-points or iteration [Has12].

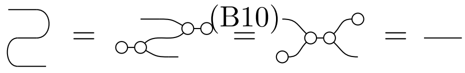
Definition 5.1.22. The *cup* and *cap* are defined as the diagrams  and , respectively.

The cup and cap satisfy the 'snake' equations thus consist a compact closed structure, whose (simple) proof we present below for integrity. The snake equation intuitively says that one can pull straight any bent wire.

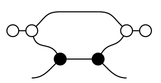
Proposition 5.1.23. *The following 'snake equations' hold for our definition of cup and cap:*

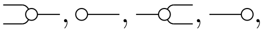


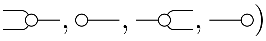
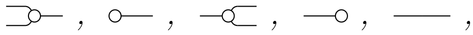
Proof. Below we only prove one of the equations, as the other one is completely symmetric.

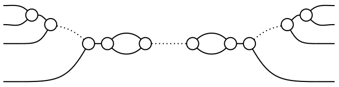


□

Note that axiom (A14) uses the syntactic sugar and , namely its lhs is .

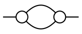
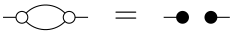
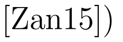
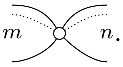
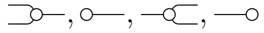
Moreover, Frobenius algebras greatly reduce the mental load in keeping track of different ways of composing , since (B9)-(B10) intuitively says that diagrams are equivalent up to topological connectedness. This is formally stated as the *spider theorem* — see e.g. [HV19, Theorem 5.21].

Proposition 5.1.24 (Spider theorem). *If (**) forms a commutative Frobenius structure, then any* connected *string diagram made out of*  *, using composition and the monoidal product, is equal to the following normal form*

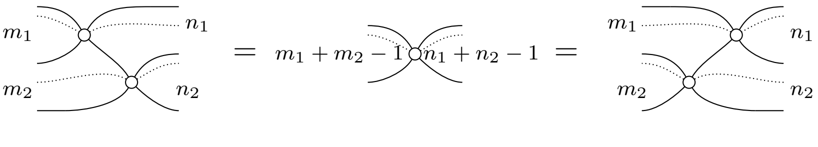


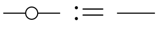
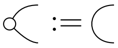
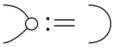
(5.3)

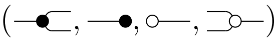
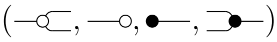
Here the word 'connected' refers to the diagram as a undirected graph: a diagram is connected if there exists at least one path between any two white nodes in the diagram.

In our axiomatic theory, we can simplify the normal form of Proposition 5.1.24 even further to get rid of those  in (5.3). Making crucial use of axiom (A14), we can derive  (5.2.5) (see Corollary 5.2.5). Note it is unusual for a Frobenius algebra to satisfy (5.2.5) as those that have appeared in the literature satisfy a version of (5.2.5) with the rhs set to the identity (e.g. [Zan15]) or to  (e.g. [CK10]). Thanks to the spider theorem and equation (∗), we can see any *simply connected* composition of white monoids and comonoids as a single node, whose only relevant structure is its number of dangling wires on the left and on the right. The normal form guarantees that we can unambiguously depict any such morphism $m \to n$ as a *spider* with m wires on the left and n on the right: . With this syntactic sugar, the axioms of the Frobenius structure  can be concisely expressed via the following *spider fusion* scheme.

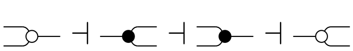
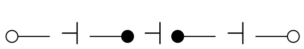
Proposition 5.1.25. *The following axiom scheme is equivalent to (B1)-(B10),*

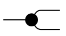
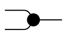
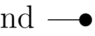
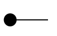


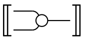
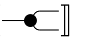
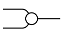
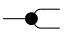
where    *, and* $m_1 + m_2, n_1 + n_2 \geq 1$.

Block C axioms. (C1)-(C4) and (C5)-(C8) state that  and  also form two bimonoids, respectively. Semantically, these hold when the underlying poset is a lattice, namely it has binary meets and joins. They also imply that $\mathbb{SATA}$ contains $\mathsf{Mat}\ \mathbb{B}$ as a monoidal subcategory, the category of Boolean matrices with the direct sum as monoidal product (*cf.* Section 5.3).

Block D axioms. The D axioms spell out a number of adjunctions in the 2-categorical sense [JY21]. The situation can be summarised by the following six adjunctions:

The adjunctions  ⊣  and  ⊣  give the defining inequations of cartesian bicategories [CW87].

To understand where the other adjunctions come from, it is helpful to adopt a semantic perspective. For example, recall that ⟦⟧ = $\{((x_1, x_2), y) \mid x_1 \wedge x_2 \leq y\}$ and ⟦⟧ = $\{(x, (y_1, y_2)) \mid x \leq y_1, x \leq y_2\} = \{(x, (y_1, y_2)) \mid x \leq y_1 \wedge y_2\}$. Thus, one can see the adjunction  ⊣  as arising from the duality between the two different ways of turning the monotone function $\wedge : \mathbb{B}^2 \to \mathbb{B}$ into a monotone relation (by placing it on either side of the $\leq$ symbol in the corresponding set comprehension).

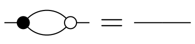
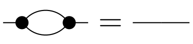
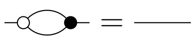
Semantically, they guarantee that the underlying poset $\mathbb{B}$ is a *lattice*, *i.e.*, has binary meets and joins. Together with the Frobenius axioms (B9)-(B10), they imply that $\mathbb{B}$ is a complemented distributive lattice, in other words, a Boolean algebra.

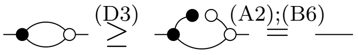
5.2 Some useful results in *SATA*

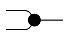
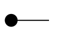
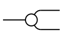
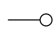
In this subsection we list a few useful (in)equations derivable from the axioms in *SATA*. While most of them will be used for proving completeness (Section 5.4, Section 5.5), some of them are handy and highlight the difference between *SATA* and GLA.

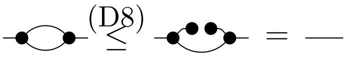
First we point out that for some of the inequalities in block D of Figure 5.1, their converse also hold thus are indeed equalities. We list and prove all of them below.

Proposition 5.2.1. *The converse inequalities to (D2), (D5), and (D10) are derivable, thus:*

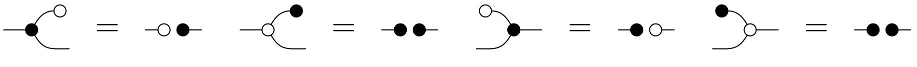
 (D2')  (D5')  (D10')

Proof. For (D2) we can derive the converse inequality as follows: 

. The converse inequality of (D10) is essentially reflecting each diagram in the above proof, where one replaces (D3) by (D7) and applys unitality (A6) of ,  and counitality (B2) of , .

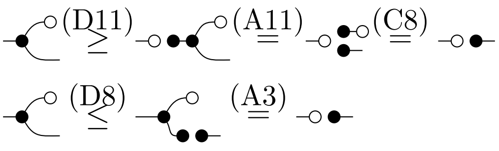
For (D5),  . □

Next we have a look at how black/white monoids (resp. comonoids) interact with white/black units (resp. counits).

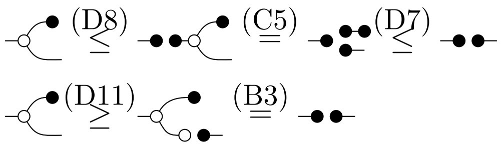
Proposition 5.2.2. 

Proof. We only prove the first two equations, and the other two follow from a similar proof by reflecting all the diagrams in the proof (note that all the axioms in *SATA* has its mirror-image counterpart).

For the first equation,



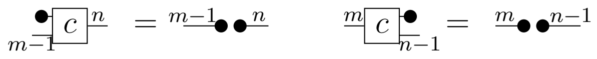
For the second equation,



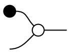
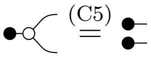
□

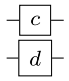
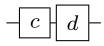
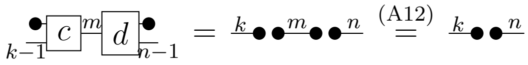
Moreover, Proposition 5.2.2 entails that —● and ●— 'disconnect' ○-structure.

Proposition 5.2.3. *For any diagram $c\colon m \to n$ freely composed of* ⊐○—, ○—, —○⊏, —○, *if c is connected, then the following equations hold whenever the numbers involved are all natural numbers:*



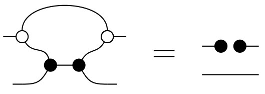
(5.4)

Proof. For simplicity we prove the first equation only; the second follows from a proof that essentially reflects all the diagrams in the proof below. We prove by induction on the structure of diagram c. For the base case, we have: ●—○ = ⬚ for c = —○;  = —●●— for c = ⊐○— by Proposition 5.2.2;  for c = ⊐○—; trivial for c = ——; not applicable to c = ○—.

For the induction step, we show that parallel and sequential composition preserve this property. For  , this diagram is not connected unless (at least) one of c and d is ⬚, in which case the statement holds trivially via induction hypothesis. For , suppose in addition that they have types $c\colon k \to m$ and $d\colon m \to n$, then , thus also holds for c ; d. □

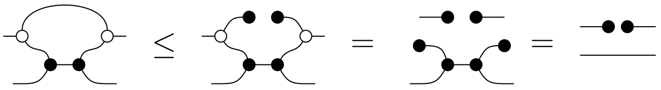
The next proposition deals with an important case in the completeness proof (Section 5.5) in which one may have 'loops' (*i.e.* non-terminating process) when rewriting a diagram into normal form.

Proposition 5.2.4.



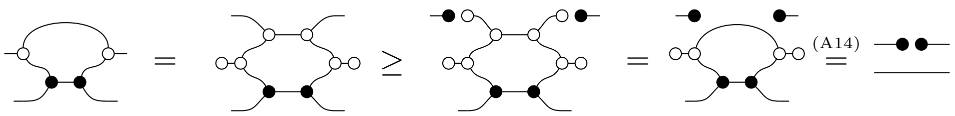
(loop)

Proof.



(5.5)

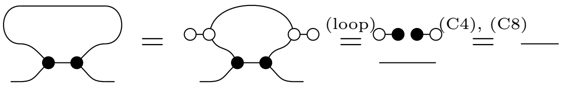
Crucially we need (A14) for the second inequality:



(5.6)

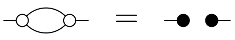
□

In fact, (loop) is equivalent to (A14) given all the *SATA* axioms except (A14):

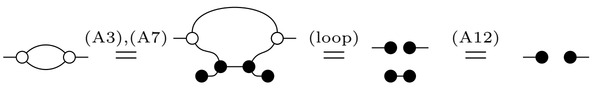


Apart from its importance in the completeness proof, we also need (loop) to prove the following property of the Forbenius structure in *SATA*.

Corollary 5.2.5.

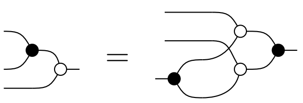


Proof.



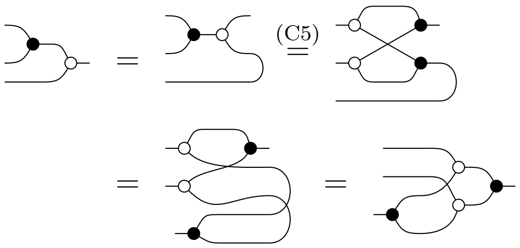
□

Proposition 5.2.6. *The following distributivity equation holds:*



(dst1)

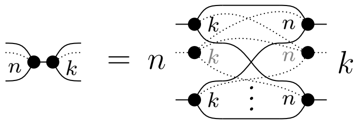
Proof.



(5.7)

□

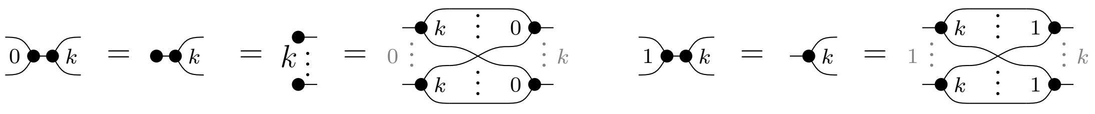
Proposition 5.2.7. *The following general bialgebra equation holds for all natural numbers k, n.*



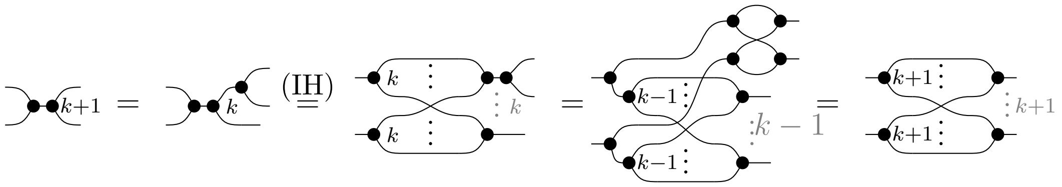
(5.8)

Proof. For readability, in the rest of the proof we will not use the dashed wires but instead use $\vdots$ and natural numbers to intuitively express that 'there are k duplicates of certain pattern'.

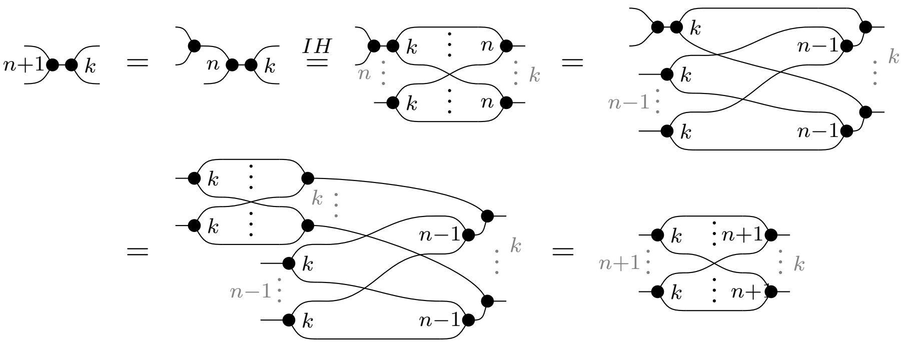
We prove by induction on n, and leave k arbitrary. When $n = 0$ and $n = 1$, we have



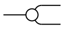
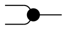
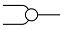
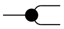
When $n = 2$, we make an induction on k. The base case for $k = 0, 1$ are exactly the same as the previous proof for $n = 0, 1$. For $k = 2$, this is exactly the bialgebra axiom. Now suppose it holds for $k \geq 2$, and we consider $k+1$:

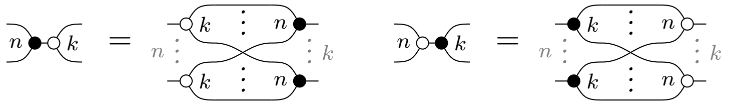


We continue with the induction proof on n. Suppose equation (5.8) holds for n, then for $n+1$,

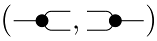
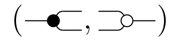
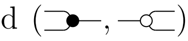


□

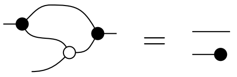
Proposition 5.2.8. *The following bialgebra equation between*  *and*  *(resp.*  *and* *) holds for arbitrary natural numbers k, n*



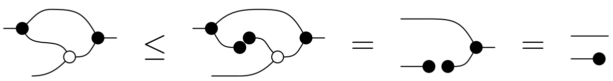
(5.9)

Proof. The proof is the same as that for Proposition 5.2.7. Note that there only properties of  forms a bimonoid is used, which also holds for  and , see the B and C axioms in Figure (5.1). □

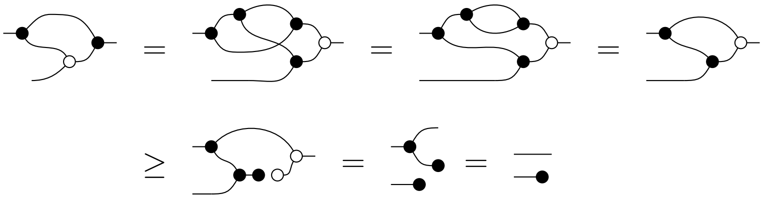
The next statement is a diagrammatic counterpart of $x_1 \leq y \;\&\; x_1 \wedge x_2 \leq y \iff x_1 \leq y$, or equivalently, $x_1 \wedge x_2 \leq x_1$.

Proposition 5.2.9. 

Proof. We prove the two inequalities. On one hand,

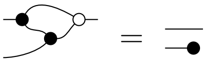


On the other hand,

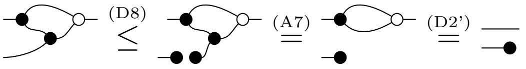


□

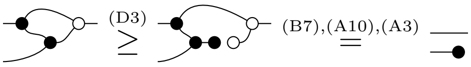
Dually, the following statement diagrammatically expresses that $x_1 \leq x_1 \vee x_2$.

Proposition 5.2.10. 

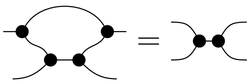
Proof. We prove the two inequations separately. On one hand,

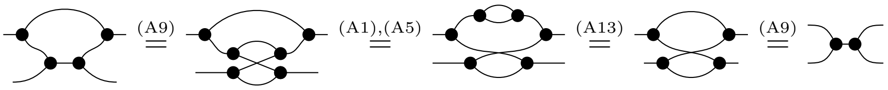


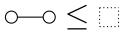
On the other hand,

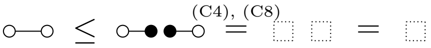


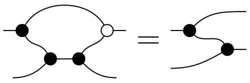
□

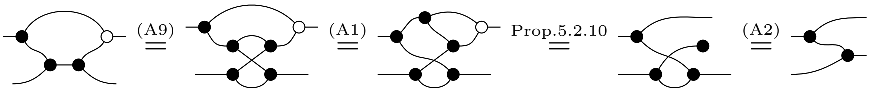
Proposition 5.2.11. 

Proof. 
□

Proposition 5.2.12. 

Proof. 
□

Proposition 5.2.13. 

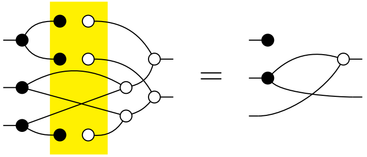
Proof. 
□

5.3 Matrix diagrams

In this subsection we take a necessary detour to Boolean matrices and matrix diagrams. Matrix diagrams are fundamental components of graphical linear algebra (see Section 2.3.1). In a nutshell, they are diagrams that encode matrices, and the axioms for (the generators in) matrix diagrams guarantee that such expression is full and faithful, thus called a *presentation*.

In our case, the theory *SATA* and the interpretation of diagrams under $[\![\cdot]\!]$ are essentially *inequational*, so the connection between the diagrams and the semantic objects is non-standard, and deserves further exploration. In a nutshell, SATA is expressive enough to express all monotone relations between finite Boolean algebras, and the latter is the appropriate context to study semantics for propositional definite logic programs. Even though technically such connection between the diagrammatic language SATA and monotone relations can be established without introducing other concepts, in order to make the intuition clearer, we use Boolean matrices as the bridge between (the (⊐●—, —○⊏)-fragment of) SATA and those monotone relations that represent monotone functions.

Example 5.3.1. Before entering the technical development, we provide some intuition of matrix diagrams and their connection with Boolean matrices in our framework. In *SATA*, the 2×3 Boolean matrix $A = \begin{pmatrix} 0 & 1 & 1 \\ 0 & 1 & 0 \end{pmatrix}$ will be represented diagrammatically as the following diagram $d_A \colon 3 \to 2$, say:

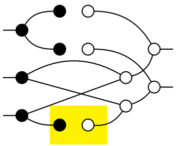


(5.10)

This is because semantically, matrix A represents the coefficients of a set of inequalities whose set of solution is exactly the interpretation of d_A:

$$\begin{aligned} [\![d_A]\!] &= \{((x_1,x_2,x_3),(y_1,y_2)) \mid x_2 \wedge x_3 \leq y_1 \;\&\; x_2 \leq y_2, x_1,x_2,x_3,y_1,y_2 \in \mathbb{B}\} \\ &= \{((x_1,x_2,x_3),(y_1,y_2)) \mid A \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \leq \begin{pmatrix} y_1 \\ y_2 \end{pmatrix}\} \end{aligned}$$

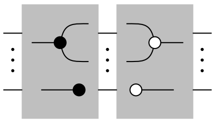
Moreover, taking a closer look at the left-hand-side of (5.10), we notice a syntactic correspondence between the highlighted part of the diagram and the entries of matrix A: $A_{ij} = 1$ precisely when the j-th wire on the left port is connected with the i-th wire on the right port via —— ; $A_{ij} = 0$ precisely if the j-th wire on the left port is disconnected with the i-th wire on the right port via —●○— . For instance, the highlighted part in



corresponds to $A_{23} = 0$. The next step is then to make a formal account of such intuitive notion of (dis)connection and the correspondence between the diagrammatic components and matrix entries, resulting in Definition 5.3.5, Proposition 5.3.11.

What underlies matrix diagrams is commutative bimonoid structure. Indeed there are three commutative bimonoid structures in *SATA*, which we refer to as (—●⊂, ⊃●—), (—●⊂, ⊃○—), and (—○⊂, ⊃●—). From a syntactic point of view, the bimonoid axioms entail that the monoid structure always 'traverses through' the comonoid structure. For example, (C6) says that when ⊃●— meets —○⊂, the ⊃●— can pass through the —○⊂ from left to right by duplicating both ⊃●— and —○⊂, and then connect in a certain way. Formally, this means that diagrams in a bimonoidal structure always have a canonical form such that the comonoid structure appear before the monoid structure. We call such canonical form *matrix diagrams* (see Definition 5.3.2). Below we state the definition and properties of matrix diagrams for the bimonoid structure (—●⊂, ⊃○—) since it will be explicitly used in the completeness proof in Section 5.4 and Section 5.5. The similar results for (—●⊂, ⊃●—) and (—○⊂, ⊃●—) can be obtained by replacing —●⊂ and ⊃○— in the following definitions and proofs with the chosen (co)monoid structures. While they do not play an important role in the completeness proof, we list the essential details in Remark 5.3.13.

Definition 5.3.2. A Syn-morphism d is a *matrix diagram* if it factorises as follows:



(5.11)

In words, a matrix diagram d is the composite of two parts $d = d_1 ; d_2$, where d_1 contains only —●⊂ and —●, while d_2 is solely composed of ⊃○— and ○—.

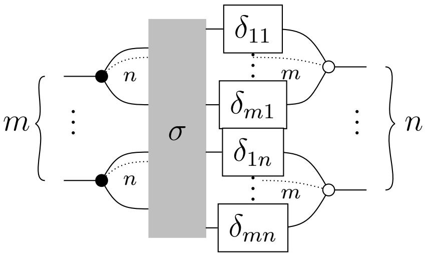
Proposition 5.3.3. *Every diagram composed of* —●⊂ , ●— , ⊃○— , ○— *is equivalent to a matrix diagram.*

Proof. This is a standard result, for example Bonchi et al. [BSZ17]. □

From a semantic point of view, under the interpretation $[\![\cdot]\!]$, matrix diagrams denote relations that are the satisfying assignments of finite sets of inequalities (over Boolean variables) of the form $x_1 \wedge \cdots \wedge x_k \leq y$. Equivalently, in standard logic terms, matrix diagrams denote finite sets of Horn clauses, *i.e.* clauses with at most one positive (unnegated) literal. More specifically, for each Horn clause $\neg x_1 \vee \cdots \vee \neg x_k \vee y$, the negated literals $\neg x_1, \ldots, \neg x_k$ correspond to a subset of the left wires, while the only positive literal y corresponds to the right wire to which those left wires are connected. For this reason, matrix diagrams can be thought of as matrices with Boolean coefficients: each row encodes the negative literals of a Horn clause through its coefficients.

We formalise this idea by explaining how every Boolean matrix is represented by a matrix diagram, and vice versa. For this we need to formalise the intuitive notion of 'being connected', as mentioned in Example 5.3.1. While the notion of '(dis)connected' is fairly intuitive, for a formal and concise definition we find it handful to introduce the canonical form of matrix diagrams.

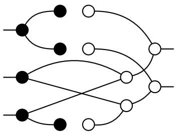
Definition 5.3.4. A *canonical matrix diagram* $d \colon m \to n$ is a matrix diagram of the following form:



(5.12)

where each δ_{ij} is either —— or —●○—, for $i = 1, \ldots, m$ and $j = 1, \ldots, n$; σ contains suitably many ⊃⊂ that move the j-th right wire of the i-th —●⊂n to the i-th left wire of the j-th m⊃○—.

Definition 5.3.5. Let $d \colon m \to n$ be a canonical matrix diagram as in Definition 5.3.4. The i-th wire on the left port and the j-th wire on the right port are *connected* if $\delta_{ij} =$ ——. Two wires on the left port, say the i-th and the i'-th one, are *connected* if they are connected to a same wire on the right, namely if there exists $j \in \{1, \dots, n\}$ such that $\delta_{ij} = \delta_{i'j} =$ ——.

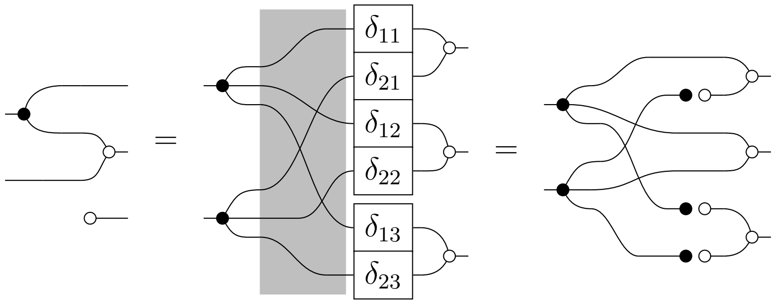
Definition 5.3.6. Recall the diagram  from Example 5.3.1, which is indeed a matrix diagram of canonical form. Here for instance, the second wire on the left port is connected to the first wire on the right port, namely $\delta_{21} =$ ——; the third wire on the left port is not connected to the second wire on the right port, thus $\delta_{23} =$ —● ○—.

Although Definition 5.3.5 only considers connectedness in canonical matrix diagram, it generalises to all matrix diagrams by observing that every matrix diagram is equivalent to a canonical one.

Corollary 5.3.7. *Any diagram formed only of the generators* —●<, —●, ○—, >○— *is equal to a canonical matrix diagram.*

Proof. It follows from Proposition 5.3.3 and the unital axioms (A2), (A3), (B6), (B7), which enable one to construct those δ_{ij} accordingly. □

Example 5.3.8. The following leftmost matrix diagram is equivalent to a canonical matrix diagram on the right hand side:



(5.13)

In particular, the grey diagram is the σ in Definition 5.3.4; $\delta_{21} = \delta_{13} = \delta_{23} =$ —● ○—, while the rest δ_{ij}'s are all ——.

Now we have formalised the concepts necessary for establishing the connection between Boolean matrices and matrix diagrams from Example 5.3.1. Let us state this correspondence for general matrix diagram and Boolean-value matrices. Note that this is can be seen as a purely syntactic correspondence result as neither the statement nor the proof require any semantic interpretation of the diagrams.

Lemma 5.3.9. *There is a 1-1 correspondence between matrix diagrams (up to SATA-equivalence) and Boolean matrices.*

Proof. The symmetric monoidal category $\mathsf{Mat}\ \mathbb{B}$ of matrices with Boolean coefficients and the direct sum as monoidal product has a well-known complete axiomatisation: the theory of a commutative and idempotent bimonoid. This result can be found in Zanasi [Zan15, Proposition 3.9], where a complete axiomatisation of the symmetric monoidal category of matrices over a principal ideal domain (again with the direct sum as monoidal product) is provided. However the proof there does not make any use of additive inverses and can be adapted without changes to the case of an arbitrary semiring, like the Booleans (see Piedeleu [Pie18] for more details).

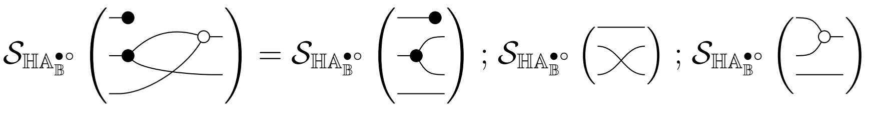
In the case of *SATA*, the equations defining a commutative and idempotent bimonoid (—●<, >○—) are given by axioms (A1)-(A4), (B5)-(B8), (C1)-(C4), and (D2') from Proposition 5.2.1. We follow the convention in [Zan15] and denote this category of SMC which is freely generated by —●<, —●, >○—, ○— and then quotiented by the theory of commutative and idempotent bimonoid as $\mathbb{HA}_{\mathbb{B}}^{\bullet\circ}$, the category of Hopf algebra (Definition 2.3.2), whose superscript indicate that it is generated by a $\bullet$-comonoid and a $\circ$-monoid. The statement of the lemma then amounts to the isomorphism between categories $\mathbb{HA}_{\mathbb{B}}^{\bullet\circ} \cong \mathsf{Mat}\ \mathbb{B}$. Then following Zanasi [Zan15], $\mathbb{HA}_{\mathbb{B}}^{\bullet\circ}$ presents $\mathsf{Mat}\ \mathbb{B}$, and the isomorphism $\mathbb{HA}_{\mathbb{B}}^{\bullet\circ} \cong \mathsf{Mat}\ \mathbb{B}$ is witnessed by the functor $\mathcal{S}_{\mathbb{HA}_{\mathbb{B}}^{\bullet\circ}}$, defined inductively on the free structure of $\mathbb{HA}_{\mathbb{B}}^{\bullet\circ}$ as follows:

$$\begin{aligned} \mathcal{S}_{\mathbb{HA}_{\mathbb{B}}^{\bullet\circ}}(\text{—●<}) &= \begin{pmatrix} 1 \\ 1 \end{pmatrix} & \qquad \mathcal{S}_{\mathbb{HA}_{\mathbb{B}}^{\bullet\circ}}(\text{—●}) &= \text{¡} \\ \mathcal{S}_{\mathbb{HA}_{\mathbb{B}}^{\bullet\circ}}(\text{>○—}) &= \begin{pmatrix} 1 & 1 \end{pmatrix} & \qquad \mathcal{S}_{\mathbb{HA}_{\mathbb{B}}^{\bullet\circ}}(\text{○—}) &= \,! \end{aligned}$$

where ¡ and ! are the universal map given by the final and initial objects in $\mathsf{Mat}\ \mathbb{B}$, respectively. As for the swapping morphism ⋈, one has $\mathcal{S}_{\mathbb{HA}_{\mathbb{B}}^{\bullet\circ}}(\text{⋈}) = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$ □

Example 5.3.10. It is helpful to see how the functor $\mathcal{S}_{\mathbb{HA}_{\mathbb{B}}^{\bullet\circ}}$ in the previous proof works on matrix diagrams, and that it fits the intuition about the translation between matrix diagrams and Boolean matrices.

Consider the matrix diagram d_A from Example 5.3.1. Applying $\mathcal{S}_{\mathbb{HA}_{\mathbb{B}}^{\bullet\circ}}$ to d_A, we have:



$$\begin{aligned} &= (\mathcal{S}_{\mathbb{HA}_{\mathbb{B}}^{\bullet\circ}}(\text{>○—}) \otimes \mathcal{S}_{\mathbb{HA}_{\mathbb{B}}^{\bullet\circ}}(\text{——})) \circ (\mathcal{S}_{\mathbb{HA}_{\mathbb{B}}^{\bullet\circ}}(\text{——}) \otimes \mathcal{S}_{\mathbb{HA}_{\mathbb{B}}^{\bullet\circ}}(\text{⋈}))\circ \\ &\qquad (\mathcal{S}_{\mathbb{HA}_{\mathbb{B}}^{\bullet\circ}}(\text{—●}) \otimes \mathcal{S}_{\mathbb{HA}_{\mathbb{B}}^{\bullet\circ}}(\text{—●<}) \otimes \mathcal{S}_{\mathbb{HA}_{\mathbb{B}}^{\bullet\circ}}(\text{——})) \\ &= \begin{pmatrix} 1 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \\ &= \begin{pmatrix} 0 & 1 & 1 \\ 0 & 1 & 0 \end{pmatrix} = A \end{aligned}$$

To our purpose, this 1-1 correspondence between matrix diagrams and Boolean matrices is an essential step in showing the completeness result of matrix diagrams.

Proposition 5.3.11 (Matrix completeness)**.** *Let $c, d\colon m \to n$ be two diagrams formed only of the generators* —●⊂, —●, ○—, ⊃○—. *We have* $[\![c]\!] = [\![d]\!]$ *iff* $c =_{SATA} d$.

Proof. The 'if' direction is straightforward, indeed follows from the soundness of $[\![\cdot]\!]$.

The 'only if' direction amounts to saying that the functor $[\![\cdot]\!]$ restricted to $\mathbb{HA}^{\mathbb{B}}_{\bullet\circ}$ is faithful. The proof strategy can be summarised by the following diagram:

$$\begin{tikzcd}
\mathbb{HA}_{\mathbb{B}}^{\bullet\circ} \arrow[r, "\mathcal{S}_{\mathbb{HA}_{\mathbb{B}}^{\bullet\circ}}"] \arrow[rrr, bend right, "{[\![\cdot]\!]}"'] & \mathsf{Mat}\ \mathbb{B} \arrow[r, "\mathcal{I}"] & \mathsf{MonFun}_{\mathbb{B}} \arrow[r, "\mathcal{R}"] & \mathsf{MonRel}_{\mathbb{B}}
\end{tikzcd} \tag{5.14}$$

In words, we divide the task of proving the faithfulness of $[\![\cdot]\!]$ into showing that the three functors are all faithful.

First, the functor $\mathcal{S}_{\mathbb{HA}_{\mathbb{B}}^{\bullet\circ}}$ from Lemma 5.3.9 is indeed an isomorphism. Second, the functor $\mathcal{R}$ representing monotone function over Booleans as monotone relations is also faithful. Third, we show that that $\mathsf{Mat}\ \mathbb{B}$ embeds faithfully in $\mathsf{MonFun}_{\mathbb{B}}$ through the following functor $\mathcal{I}$, defined inductively on the free structure of $\mathsf{Mat}\ \mathbb{B}$:

- on objects, $\mathcal{I}(n) := \mathbb{B}^n$ for arbitrary natural number n;
- on morphisms, given a $n \times m$ Boolean matrix A, $\mathcal{I}(A)\colon \mathbb{B}^m \to \mathbb{B}^n$ maps $(b_1, \ldots, b_m)$ to $(c_1, \ldots, c_n)$ where $c_i = (A_{i1} \wedge b_1) \wedge \cdots \wedge (A_{im} \wedge b_m)$.

In other words, $\mathcal{I}(A)$ is a monotone function whose j-th component $\mathcal{I}(A)_j$ (for $j = 1, \ldots, n$) is given by $\mathcal{I}(A)_j(\mathbf{b}) = b_{i_1} \wedge \cdots \wedge b_{i_k}$ where $\{i_1, \ldots, i_k\}$ is the set of indices in the i-th row of A which are equal to 1. For instance, $\mathcal{I}$ maps the matrix $A = \begin{pmatrix} 0 & 1 & 1 \\ 0 & 1 & 0 \end{pmatrix}$ from Example 5.3.1 to the function $\lambda(x_1 x_2 x_3).(x_2 \wedge x_3, x_3)$. Clearly, any two matrices that define the same monotone function are equal, thus $\mathcal{I}$ is faithful.

Finally we need to verify that (5.14) commutes. This can be shown inductively on the structure of $\mathbb{HA}^{\mathbb{B}}_{\bullet\circ}$. For the inductive base, it holds for all generating morphisms:

- $(\mathcal{R} \circ \mathcal{I} \circ \mathcal{S}_{\mathbb{HA}_{\mathbb{B}}^{\bullet\circ}})($—●⊂$) = (\mathcal{R} \circ \mathcal{I})(\begin{pmatrix} 1 \\ 1 \end{pmatrix}) = \mathcal{R}(\lambda x.(x, x)) = \{(x, (y_1, y_2) \mid x \leq y_1, y_2)\} = [\![$—●⊂$]\!]$
- $(\mathcal{R} \circ \mathcal{I} \circ \mathcal{S}_{\mathbb{HA}_{\mathbb{B}}^{\bullet\circ}})($—●$) = (\mathcal{R} \circ \mathcal{I})(¡) = \mathcal{R}(\lambda x.\bullet) = \{(x, \bullet) \mid x \in \mathbb{B}\} = [\![$—●$]\!]$
- $(\mathcal{R} \circ \mathcal{I} \circ \mathcal{S}_{\mathbb{HA}_{\mathbb{B}}^{\bullet\circ}})($⊃○—$) = (\mathcal{R} \circ \mathcal{I})(\begin{pmatrix} 1 & 1 \end{pmatrix}) = \mathcal{R}(\lambda(x_1, x_2).x_1 \wedge x_2) = \{((x_1, x_2), y) \mid x_1 \wedge x_2 \leq y\} = [\![$⊃○—$]\!]$
- $(\mathcal{R} \circ \mathcal{I} \circ \mathcal{S}_{\mathbb{HA}_{\mathbb{B}}^{\bullet\circ}})($○—$) = (\mathcal{R} \circ \mathcal{I})(!) = \mathcal{R}(\bullet \mapsto 1) = \{(\bullet, 1)\} = [\![$○—$]\!]$

For the induction step, the statement follows by noticing that $\mathcal{S}_{\mathbb{HA}_{\mathbb{B}}^{\bullet\circ}}, \mathcal{I}, \mathcal{R}, [\![\cdot]\!]$ are all SMC functors.

Now, since $[\![\cdot]\!]$ can be decomposed as the sequential composition of three faithful functors, the functor $[\![\cdot]\!]$ itself is also faithful. This means that for any c, d, if $[\![c]\!] = [\![d]\!]$ then $c =_{SATA} d$. □

Below is a handy result describing the semantic connection between matrix diagrams and their corresponding Boolean matrices.

Proposition 5.3.12. *Given an arbitrary matrix diagram $d\colon m \to n$ and suppose its corresponding Boolean matrix is A (*i.e. *$\mathcal{S}_{\mathbb{HA}^{\bullet\circ}_{\mathbb{B}}}(d) = A$). Then for all $x_1, \ldots, x_m, y_1, \ldots, y_n \in \mathbb{B}$,*

$$(\mathbf{x}, \mathbf{y}) \in [\![d]\!] \iff A \begin{pmatrix} x_1 \\ \vdots \\ x_m \end{pmatrix} \leq \begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix}$$

where the Boolean matrix multiplication is under $\langle \mathbb{B}, \vee, \wedge \rangle$.

Proof. Without loss of generality we assume that d is a canonical matrix diagram, and follow the notation from Definition 5.3.4. Suppose for each $j \in \{1, \ldots, n\}$, $\{i^j_1, \ldots, i^j_{r_j}\}$ is the set of all the indices $i \in \{1, \ldots, m\}$ such that $A_{ji} = 1$. Then by construction, $i^j_1, \ldots, i^j_{r_j}$ are also exactly the indices i such that $\delta_{ij} =$ ——.

For arbitrary $x_1, \ldots, x_m, y_1, \ldots, y_n \in \mathbb{B}$, $(\mathbf{x}, \mathbf{y}) \in [\![d]\!]$ if and only if $x_{i^j_1} \wedge \cdots \wedge x_{i^j_{r_j}} \leq y_j$ for all $j \in \{1, \ldots, n\}$. This is exactly saying that for the i-th row $A_{i\_}$ of A, $A_{i\_}(x_1, \ldots, x_m)^\top \leq y_j$, for all $j = 1, \ldots, n$. Thus it is equivalent to $A(x_1, \ldots, x_m)^\top \leq (y_1, \ldots, y_n)^\top$. □

Proposition 5.3.11 will greatly simplify our reasoning in the proof of completeness (Subsection 5.4). In light of Proposition 5.3.11, we will assume — wherever it is convenient — that we can always rewrite any diagram formed of —●⊂, —●, ○—, ⊃○— to its equivalent (canonical) matrix form, while keeping the semantics unchanged. This often means that we can be as lax as necessary when identifying diagrams modulo the (co)associativity of —●⊂ or ⊃○— and the (co)unitality of (—●⊂, —●) and (⊃○—, ○—).

Remark 5.3.13. As we have mentioned at the beginning of this subsection, the equational theory of Fig. 5.1 contains three commutative and idempotent bimonoids: (—●⊂, ⊃○—) (as we have already explained), (—●⊂, ⊃●—) and (—○⊂, ⊃●—). Unsurprisingly, completeness results similar to Proposition 5.3.11 also hold for (—●⊂, ⊃●—) and (—○⊂, ⊃●—) matrix diagrams, which can be proved using the same line of thoughts as the proof for Proposition 5.3.11 by modifying (5.14). We give the sketch below.

(—●⊂, ⊃●—)-matrix diagrams. The equations of commutative idempotent bimonoids consist of (A1)-(A13). We define the category $\mathbb{HA}^{\bullet\bullet}_{\mathbb{B}}$ as the freely generated SMC quotiented by the theory of commutative idempotent bimonoid. Then we have a similar decomposition of the functor $[\![\cdot]\!]$ restricted to $\mathbb{HA}^{\bullet\bullet}_{\mathbb{B}}$:

$$\begin{tikzcd} \mathbb{HA}^{\bullet\bullet}_{\mathbb{B}} \arrow[r, "\mathcal{S}_{\mathbb{HA}^{\bullet\bullet}_{\mathbb{B}}}"] \arrow[rrr, bend right, "{[\![\cdot]\!]}"'] & \mathsf{Mat}\ \mathbb{B} \arrow[r, "\mathcal{J}"] & \mathsf{MonFun}_{\mathbb{B}} \arrow[r, "\mathcal{R}"] & \mathsf{MonRel}_{\mathbb{B}} \end{tikzcd} \tag{5.15}$$

Here $\mathcal{S}_{\mathbb{HA}_{\mathbb{B}}^{\bullet\bullet}}$ is inductively defined as follows, and witnesses the isomorphism $\mathbb{HA}_{\mathbb{B}}^{\bullet\bullet} \cong \mathsf{Mat}\ \mathbb{B}$:

$$
\begin{aligned}
\mathcal{S}_{\mathbb{HA}_{\mathbb{B}}^{\bullet\bullet}}(\text{—●<}) &= \begin{pmatrix}1\\1\end{pmatrix} &\qquad \mathcal{S}_{\mathbb{HA}_{\mathbb{B}}^{\bullet\bullet}}(\text{—●}) &= ¡\\
\mathcal{S}_{\mathbb{HA}_{\mathbb{B}}^{\bullet\bullet}}(\text{>●—}) &= \begin{pmatrix}1 & 1\end{pmatrix} &\qquad \mathcal{S}_{\mathbb{HA}_{\mathbb{B}}^{\bullet\bullet}}(\text{●—}) &= !
\end{aligned}
$$

Moreover, $\mathcal{J}$ is defined as:

- on objects, $\mathcal{J}(n) = \mathbb{B}^n$;
- on morphisms, given a $n \times m$ Boolean matrix A, $\mathcal{J}(A)\colon \mathbb{B}^m \to \mathbb{B}^n$ maps $(b_1, \ldots, b_m)$ to $(c_1, \ldots, c_n)$ where each $c_i = (A_{i1} \wedge b_1) \vee \cdots \vee (A_{im} \wedge b_m)$.

(—○<, >●—)-matrix diagrams. In this case we can no longer factorise $[\![\cdot]\!]$ through $\mathcal{R}\colon \mathsf{MonFun}_{\mathbb{B}} \to \mathsf{MonRel}_{\mathbb{B}}$. Instead we have the following decomposition

$$\begin{tikzcd}
\mathbb{HA}_{\mathbb{B}}^{\circ\bullet} \arrow[r, "\mathcal{S}_{\mathbb{HA}_{\mathbb{B}}^{\circ\bullet}}"] \arrow[rr, bend right, "{[\![\cdot]\!]}"'] & \mathsf{Mat}\ \mathbb{B} \arrow[r, "\mathcal{K}"] & \mathsf{MonRel}_{\mathbb{B}}
\end{tikzcd} \tag{5.16}$$

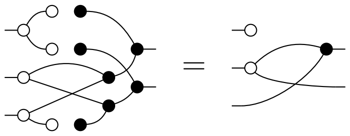
Here $\mathcal{S}_{\mathbb{HA}_{\mathbb{B}}^{\circ\bullet}}$ is inductively defined as follows, and witnesses the isomorphism $\mathbb{HA}_{\mathbb{B}}^{\circ\bullet} \cong \mathsf{Mat}\ \mathbb{B}$:

$$
\begin{aligned}
\mathcal{S}_{\mathbb{HA}_{\mathbb{B}}^{\circ\bullet}}(\text{—○<}) &= \begin{pmatrix}1\\1\end{pmatrix} &\qquad \mathcal{S}_{\mathbb{HA}_{\mathbb{B}}^{\circ\bullet}}(\text{—●}) &= ¡\\
\mathcal{S}_{\mathbb{HA}_{\mathbb{B}}^{\circ\bullet}}(\text{>●—}) &= \begin{pmatrix}1 & 1\end{pmatrix} &\qquad \mathcal{S}_{\mathbb{HA}_{\mathbb{B}}^{\circ\bullet}}(\text{○—}) &= !
\end{aligned}
$$

$\mathcal{K}$ maps a Boolean matrix to a monotone relation:

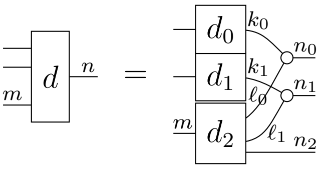
- on objects, $\mathcal{K}(n) = \mathbb{B}^n$ for arbitrary $n \in \mathbb{N}$;
- on morphisms, given a $n \times m$ Boolean matrix A, $\mathcal{K}(A)\colon \mathbb{B}^m \nrightarrow \mathbb{B}^n$ consists of all $(\mathbf{b}, \mathbf{c})$ such that for each $i \in \{1, \ldots, m\}$, $b_i \leq c_{i_1} \vee \cdots \vee c_{i_k}$ where $i_1, \ldots, i_k$ are all the row indices in the i-th column of A that has entry 1.

Example 5.3.14. Again, consider the matrix $A = \begin{pmatrix}0 & 1 & 1\\0 & 1 & 0\end{pmatrix}$ from Example 5.3.1, and the monotone relation $\mathcal{K}(A)$ consists of all $((b_1, b_2, b_3), (c_1, c_2))$ satisfying: $b_1 \leq 0$, $b_2 \leq c_1 \vee c_2$, $b_3 \leq c_1$. This is exactly the interpretation of the following diagram under $[\![\cdot]\!]$, which are the (—○<, >●—) (canonical) matrix diagrams corresponding to A:



Another useful fact for the completeness proof is that, when certain wires are not connected, a matrix diagram can be decomposed into smaller pieces of matrix diagrams. This fact will be later used in the proof of the completeness of *SATA*. In particular it makes the induction step there pass through. We state and prove below only the result for (—●⊏, ⊐○—) matrix diagrams.

Proposition 5.3.15. *Let $d\colon m+2 \to n$ be a matrix diagram (where $m, n \in \mathbb{N}$). If two wires on the left port are not connected, say the first and second wires, then d can be decomposed as follows, where d_0, d_1, d_2 are also matrix diagrams:*

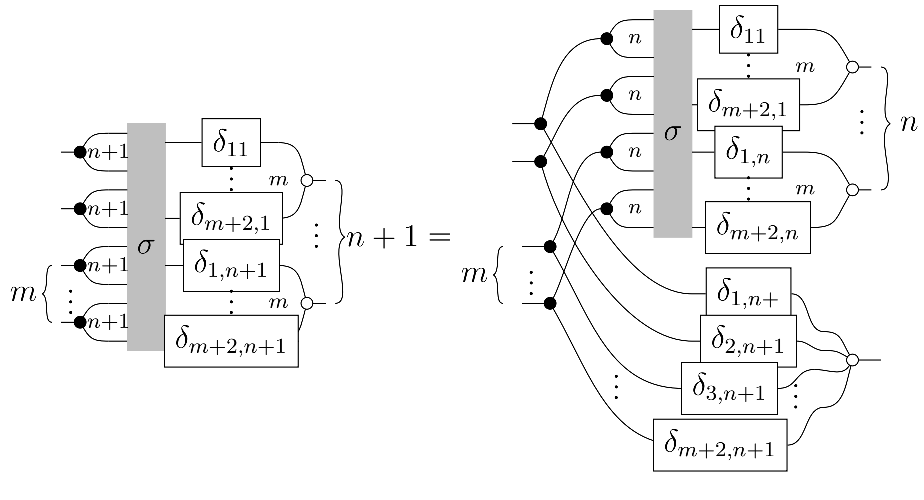


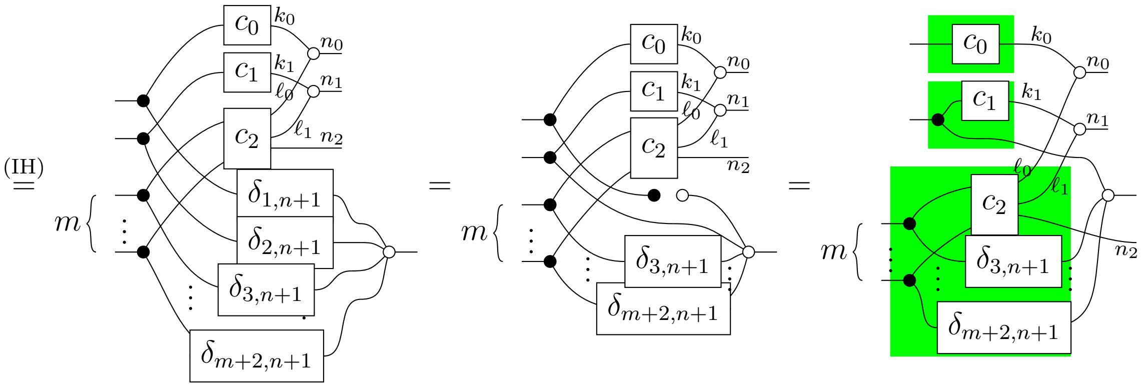
(5.17)

Proof. In this proof we will use curly brackets and natural numbers k to indicate that certain pattern appears k many times.

Without loss of generality we assume that d is a canonical matrix diagram. While one can prove directly on d for arbitrary m, n, to avoid messing up the wires we present here a proof by induction on $n \in \mathbb{N}$. If $n = 0$, then (5.17) is trivial: d is equivalent to the parallel composition of $m+2$-many —●.

Suppose (5.17) works for arbitrary matrix diagram of type $m+2 \to n$, we show that it also holds for canonical matrix diagram $d\colon m+2 \to n+1$. Since top two wires on the left port are not connected, wlog we assume that $\delta_{0n} =$ —●○— and $\delta_{0n} =$ —— (the case where both are —●○— is simpler). Then d can be decomposed as follows, where the three green highlighted (matrix) diagrams work respectively as d_0, d_1, d_2 in the rhs of (5.17):





This finishes the induction step. □

5.4 Soundness and completeness for monotone Boolean functions

Before proving the completeness result for *SATA* and $\mathsf{MonRel}_{\mathbb{B}}$, we first provide some taste of the calculus by showing the completeness for a simpler system (indeed a subsystem of *SATA*). We first look at the semantics. We will consider the functional fragment of $\mathsf{MonRel}_{\mathbb{B}}$, namely those monotone relations that are representations of some monotone functions.

Definition 5.4.1. Let $\mathsf{MonRF}_{\mathbb{B}}$ be the poset-enriched subcategory of $\mathsf{MonRel}_{\mathbb{B}}$ whose morphisms are monotone relations that are generated by some monotone functions.

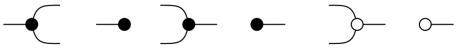
Equivalently, $\mathsf{MonRF}_{\mathbb{B}}$ is the image of the embedding functor $\mathcal{R}\colon \mathsf{MonFun}_{\mathbb{B}} \to \mathsf{MonRel}_{\mathbb{B}}$. A morphism of type $\mathbb{B}^m \nrightarrow \mathbb{B}^n$ in $\mathsf{MonRF}_{\mathbb{B}}$ is a monotone relation $R \subseteq \mathbb{B}^m \times \mathbb{B}^n$ generated by some monotone function $f\colon \mathbb{B}^m \to \mathbb{B}^n$, namely $R = \mathcal{R}(f)$. $\mathsf{MonRF}_{\mathbb{B}}$ remains as a poset-enriched category, whose poset structure on morphisms is $\subseteq$.

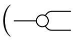
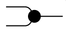
Remark 5.4.2. $\mathsf{MonRF}_{\mathbb{B}}$ is a *proper* subcategory of $\mathsf{MonRel}_{\mathbb{B}}$ because not every monotone relation is generated by a monotone function. For example, consider $R \coloneqq \{(x,(y_1,y_2)) \mid x \leq y_1 \vee y_2\}$. R is a monotone relation $\mathbb{B} \nrightarrow \mathbb{B}^2$, but there is no monotone function $f\colon \mathbb{B} \to \mathbb{B}^2$ such that $R = \mathcal{R}(f)$: for there are two minimal elements (y_1,y_2) in $\mathbb{B}^2$ such that $(1,(y_1,y_2)) \in R$, namely $(0,1)$ and $(1,0)$. Indeed R is the interpretation ⟦—○⊏⟧, and we shall soon see that it is not part of the category of string diagrams presenting (monotone relations generated by) monotone functions.

On the diagrammatic side, we consider the syntax category as the one freely generated by all the ingredients for Syn but —○⊏ and —○. Formally the syntax category Syn^{func} is defined as follows.

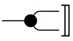
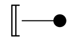
Definition 5.4.3. The category Syn^{func} is the symmetric monoidal category freely generated by the following components:

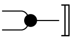
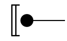
- Objects: a single generating object 1;
- Morphisms: the following generating string diagrams

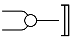
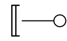


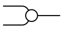
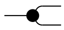
Remark 5.4.4. This explains why in Remark 5.3.13, our construction for the functor $[\![\cdot]\!]$ restricted to (, ) matrix diagram does not factorise through $\mathcal{R}$: $\mathsf{MonFun}_{\mathbb{B}} \to \mathsf{MonRel}_{\mathbb{B}}$.

It is helpful to identity which monotone functions generate the interpretations of the generators of Syn^{func} (as monotone relations).

$\mathcal{R}(\lambda x.(x, x)) = [\![$  $]\!]$ $\qquad$ $\mathcal{R}(\lambda x.\bullet) = [\![$  $]\!]$

$\mathcal{R}(\lambda x_1 x_2.x_1 \vee x_2) = [\![$  $]\!]$ $\qquad$ $\mathcal{R}(0) = [\![$  $]\!]$

$\mathcal{R}(\lambda x_1 x_2.x_1 \wedge x_2) = [\![$  $]\!]$ $\qquad$ $\mathcal{R}(1) = [\![$  $]\!]$

The inequation theory $SATA^{func}$ is the fragment of $SATA$ that only involves (in)equalities about the generating morphisms of Syn^{func}, plus the axioms stating the interaction between  and . For convenience we list them in Figure 5.2 and keep the same index of the axioms as in Figure 5.1. We quotient Syn^{func} by the inequational theory $SATA^{func}$, which again results in a poset-enriched category, called $\mathbb{SATA}^{func}$. The central theorem of this subsection is then the following soundness and completeness result, establishing the isomorphism between two poset-enriched categories.

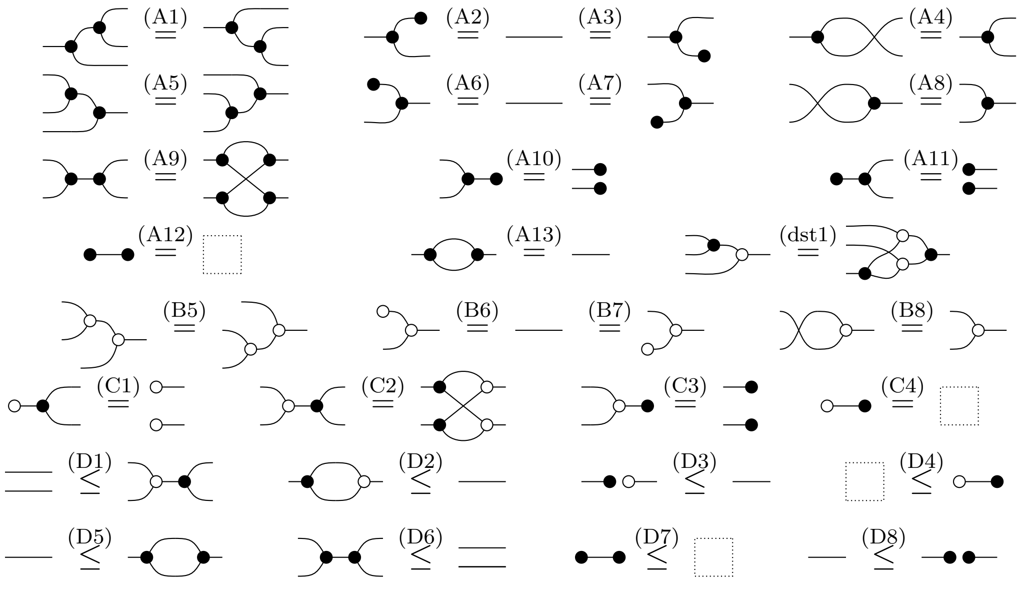


Figure 5.2: $SATA^{func}$

Theorem 5.4.5. $\mathbb{SATA}^{func} \cong \mathsf{MonRF}_{\mathbb{B}}$.

Theorem 5.4.5 amounts to showing that there is a full and faithful functor $\mathcal{F}\colon \mathbb{SATA}^{func} \to \mathsf{MonRF}_{\mathbb{B}}$ that is also bijective on objects. We have a natural candidate of this $\mathcal{F}$, namely the interpretation $[\![\cdot]\!]$ (see Definition 5.1.15) restricted to $\mathbb{SATA}^{func}$ and $\mathsf{MonRF}_{\mathbb{B}}$, which we still denote as $[\![\cdot]\!]$ with a bit abuse of notation. First, that $[\![\cdot]\!]$ is a symmetric monoidal functor follows directly from the soundness for $\mathbb{SATA}$ and $\mathsf{MonRel}_{\mathbb{B}}$ (see Proposition **??**). Next, it is immediate that $[\![\cdot]\!]$ is bijective on objects: $[\![\cdot]\!]$ maps object n in the syntax category $\mathbb{SATA}^{func}$ to object $\mathbb{B}^n$ in the semantics category $\mathsf{MonRF}_{\mathbb{B}}$, for every natural number n.

The only remaining and nontrivial task is to show that $[\![\cdot]\!]$ is full and faithful. By the definition of $\mathbb{SATA}^{func}$ as a freely generated category quotiented by certain inequational theory, these two conditions of $[\![\cdot]\!]$ boil down to the following properties:

- fullness means that for every relational representation $\mathcal{R}(f)$ of some monotone function $f\colon \mathbb{B}^m \to \mathbb{B}^n$, there exists a $\mathbb{SATA}^{func}$-morphism d such that $[\![d]\!] = \mathcal{R}(f)$ (Proposition 5.4.23);

- faithfulness says that for two $\mathbb{SATA}^{func}$-morphisms c and d, if $[\![c]\!] \subseteq [\![d]\!]$, then $c \leq_{SATA^{func}} d$ (Proposition 5.4.25).

In the following two subsections, we will prove fullness first, which shall shed some light on the pre-normal form and normal form that will be crucial in the followup faithfulness proof. One simple observation that will simplify the proof setting is that, since $\mathsf{MonFun}_{\mathbb{B}}$ is a cartesian category, so each monotone function $f\colon \mathbb{B}^m \to \mathbb{B}^n$ can be decomposed as an n-tuple $(f_1, \ldots, f_n)$ where each f_i is of type $\mathbb{B}^m \to \mathbb{B}$, and $\mathcal{R}(f) = (\mathcal{R}(f_1), \ldots, \mathcal{R}(f_n))$. Thus it suffices to restrict ourselves to $n = 1$, and show that for every monotone function $f\colon \mathbb{B}^m \to \mathbb{B}$, there exists $d\colon m \to 1$ such that $[\![d]\!] = \mathcal{R}(f)$.

Let us fix a monotone function $f\colon \mathbb{B}^m \to \mathbb{B}$. We note that f is uniquely determined by $f^{-1}(0)$: for an arbitrary $b \in \mathbb{B}^m$, $f(b) = 0$ if $b \in f^{-1}(0)$, and $f(b) = 1$ otherwise. Moreover, $f^{-1}(0)$ is always a downward closed subset of $\mathbb{B}^m$: if $f(b) = 0$ and $b' \leq b$, then by monotonicity of f, $f(b') \leq f(b) = 0$, thus $f(x') = 0$. In fact there is a one-one correspondence between monotone functions $\mathbb{B}^m \to \mathbb{B}$ and downward closed subsets of $\mathbb{B}^m$. To further exploit this observation, we will introduce the notion of $\neg$-clauses.

5.4.1 $\neg$-clauses and sets of $\neg$-clauses

In this subsection we fix a finite set of variables $\mathit{Var} = \{x_1, \ldots, x_m\}$, unless otherwise specified (such as in some examples).

Definition 5.4.6. A *$\neg$-clause* over Var is a finite subset $\{x_{i_1}, \ldots, x_{i_k}\}$ of Var. A *set of $\neg$-clauses* over Var is a finite set of $\neg$-clauses.

Definition 5.4.7. A *$\neg$-clause assignment* (or simply *assignment*) is a mapping $\mathbf{v}\colon \mathit{Var} \to \mathbb{B}$. An assignment $\mathbf{v}$ *satisfies* a $\neg$-clause $\sigma = \{x_{i_1}, \cdots, x_{i_k}\}$ if $\mathbf{v}(\neg x_{i_1}) \vee \cdots \vee \mathbf{v}(\neg x_{i_1}) = 1$, or equivalently, $\mathbf{v}(x_{i_1}) \wedge \cdots \wedge \mathbf{v}(x_{i_1}) = 0$. The *set of satisfying assignments* for σ is the set of all assignments satisfying σ, denoted as $Sat(\sigma)$. An assignment satisfies a set of $\neg$-clauses Σ if it satisfies all $\neg$-clauses in Σ. The *set of satisfying assignments* for Σ is the set of all assignments that satisfies Σ, denoted as $Sat(\Sigma)$.

One immediate observation is that, the set of satisfying assignments for a set of $\neg$-clauses is simply the intersection of the set of satisfying assignments for every $\neg$-clause in this system, namely $Sat(\Sigma) = \bigcap_{\sigma\in\Sigma} Sat(\sigma)$.

Convention 5.4.8. For convenience, we will write an assignment $\mathbf{v}\colon \{x_1, \ldots, x_m\} \to \mathbb{B}$ by a tuple $(\mathbf{v}(x_1), \ldots, \mathbf{v}(x_m))$ in $\mathbb{B}^m$, or sometimes even just by a string $\mathbf{v}(x_1)\cdots\mathbf{v}(x_m)$.

We denote $\mathbf{v}(x_{i_1}) \wedge \cdots \wedge \mathbf{v}(x_{i_1})$ by $\sigma(\mathbf{v})$, thus the condition of an assignment $\mathbf{v}$ satisfying a $\neg$-clause σ can be concisely expressed as $\sigma(\mathbf{v}) = 0$.

We use the terminology '$\neg$-clause' due to its close connection with the standard notion of clauses in logic. Recall from standard logic textbooks that a clause φ is a set of literals $\{l_1, \ldots, l_k\}$, and a satisfying assignment for φ is a Boolean assignments for atoms appearing in φ that validates φ when viewed as a disjunction, namely $l_1 \vee \cdots \vee l_k$ has truth value 1. Our notion of $\neg$-clauses and their satisfaction is the dual: a $\neg$-clause can be seen as a conjunction of its components, and satisfaction means that the conjunction has truth value 0.

Example 5.4.9. Let $Var = \{x_1, x_2, x_3\}$. Consider two $\neg$-clauses $\sigma := \{x_1, x_2\}$ and $\gamma := \{x_1, x_3\}$ be two $\neg$-clauses, and a set of $\neg$-clauses $\Sigma := \{\{x_3\}, \{x_1, x_2\}, \{x_1, x_2, x_3\}\}$. We define the assignment $\mathbf{v}_1$ to be $\mathbf{v}_1(x_1) = \mathbf{v}_1(x_3) = 0$, $\mathbf{v}_1(x_2) = 1$; or equivalently, we follow Convention 5.4.8 and write $\mathbf{v}_1 = 010$. By definition, $\mathbf{v}_1$ satisfies σ and Σ (*i.e.* $\mathbf{v}_1 \in Sat(\sigma), Sat(\Sigma)$), but not γ (*i.e.* $\mathbf{v}_1 \notin Sat(\gamma)$). The set of satisfying assignments of Σ, namely $Sat(\Sigma)$, consists of $\mathbf{v}_1$ together with other two assignments $\mathbf{v}_2 = 100$ and $\mathbf{v}_3 = 000$.

Similar to the fact that two different set of equations may have the same set of solutions, two sets of $\neg$-clauses may also share the same set of satisfying assignments. However one key observation for the latter case is that it can happen only in a naive way: one set of $\neg$-clauses is contained in the other as a subset of $\neg$-clauses. This entails that for a given set of solutions, there is always a smallest set of $\neg$-clauses whose set of solutions is this set.

Definition 5.4.10. A set of $\neg$-clauses Σ is *minimal* if for any $\Gamma \subset \Sigma$, $Sat(\Sigma) \subset Sat(\Gamma)$.

In other words, a set of $\neg$-clauses Σ is minimal if it does not have any 'redundant' $\neg$-clause, dropping which has no impact on the set of satisfying assignments.

Example 5.4.11. Recall Example 5.4.9, Σ is not minimal because its proper subset $\{\{x_3\}, \{x_1, x_2\}\}$ has the same set of satisfying assignments as Σ. In other words, the $\neg$-clause $\{x_1, x_2, x_3\}$ is redundant given either $\{x_3\}$ or $\{x_1, x_2\}$.

The next three results tell us that minimal sets of $\neg$-clauses are unique, thus *smallest*.

Lemma 5.4.12. *Consider a set of* $\neg$*-clauses* Σ *of the form* $\{\{y^1_1, \dots, y^1_{k_1}\}, \dots \{y^m_1, \dots, y^m_{k_m}\}\}$ *and a* $\neg$*-clause* $\gamma = \{x_1, \dots, x_n\}$*. If* $\mathit{Sat}(\Sigma) \subseteq \mathit{Sat}(\gamma)$*, then there exists some* $i \in \{1, \dots, m\}$ *such that* $\{y^i_1, \dots, y^i_{k_i}\} \subseteq \{x_1, \dots, x_n\}$*.*

Proof. We prove this by contradiction. Suppose the lemma does not hold, then for each $i \in \{1, \dots, m\}$, $\{y^i_1, \dots, y^i_{k_i}\} \not\subseteq \{x_1, \dots, x_n\}$, so there exists some $y^i_{p_i}$ such that $y^i_{p_i} \notin \{x_1, \dots, x_n\}$. Consider the assignment $\mathbf{v}$ such that $\mathbf{v}(y^i_{p_i}) = 0$ for all $i \in \{1, \dots, m\}$, and $\mathbf{v}(z) = 1$ for all the other variables. Then $\mathbf{v}$ satisfies Σ: for each i, $\mathbf{v}(y^i_1) \wedge \dots \wedge \mathbf{v}(y^i_{p_i}) \wedge \dots \wedge \mathbf{v}(y^i_{k_i}) = 0$. However, $\mathbf{v}$ does not satisfy γ because $\{y^1_{p_1}, \dots, y^m_{p_m}\} \cap \{x_1, \dots, x_n\} = \varnothing$ and $\mathbf{v}(x_j) = 1$ for all $j \in \{1, \dots, n\}$, thus a contradiction. □

Proposition 5.4.13. *If two sets of* $\neg$*-clauses* Σ *and* Γ *satisfy* $\mathit{Sat}(\Sigma) = \mathit{Sat}(\Gamma)$*, then* $\Sigma = \Gamma$*.*

Proof. Since the satisfying assignments of Σ form a subset of those of Γ, they are also a subset of the satisfying assignments of every $\gamma \in \Gamma$. By Lemma 5.4.12, for every $\gamma \in \Gamma$ there exists a $\sigma \in \Sigma$ such that $\sigma \subseteq \gamma$. Using a similar argument, since the satisfying assignments of Γ form a subset of those of Σ, for every $\sigma \in \Sigma$ there exists a $\gamma \in \Gamma$ such that $\gamma \subseteq \sigma$.

Now starting with an arbitrary $\gamma \in \Gamma$, there exists some $\sigma \in \Sigma$ such that $\sigma \subseteq \gamma$. For this σ there again exists some $\gamma' \in \Gamma$ such that $\gamma' \subseteq \sigma$. So $\gamma' \subseteq \gamma$, and for a minimal set of $\neg$-clauses this means $\gamma = \gamma'$: otherwise we could drop γ' and get a smaller set than Γ with the same satisfying assignments. Then $\gamma \subseteq \sigma \subseteq \gamma'$ and $\gamma = \gamma'$ imply $\gamma = \sigma = \gamma'$, which means that γ is in Σ as well. Since this holds for arbitrary $\gamma \in \Gamma$, we know that $\Gamma \subseteq \Sigma$. Applying a similar argument for Σ, we know that $\Sigma \subseteq \Gamma$. Therefore we can conclude that $\Sigma = \Gamma$. □

So far we studied enough properties of sets of $\neg$-clauses to make the connection with monotone functions of type $\mathbb{B}^m \to \mathbb{B}$, or equivalently with downward closed subsets of $\mathbb{B}^m$.

Corollary 5.4.14. *For every downward closed subset* $\mathcal{A} \subseteq \mathbb{B}^m$*, there is a unique minimal set of* $\neg$*-clauses* Σ *such that* $\mathit{Sat}(\Sigma) = \mathcal{A}$*.*

Proof. We first show the existence of such a set of $\neg$-clauses, and then it follows from Proposition 5.4.13 that there is a unique minimal one.

Note that each $b \in \mathcal{A}$ can be seen as an assignment $b \colon \mathit{Var} \to \mathbb{B}$ such that $b(i) = 0$ if and only if the i-th component of b is 0. Let Γ consists of all $\neg$-clauses σ such that $\sigma(b)$ for all $b \in \mathcal{A}$ viewed as assignments. Then by construction every $b \in \mathcal{A}$ satisfies Γ. Now given an assignment $\mathbf{v} \in \mathit{Sat}(\Gamma)$, we show that $\mathbf{v} \in \mathcal{A}$ as well, thus $\mathit{Sat}(\Gamma) = \mathcal{A}$. Suppose not, then $\mathcal{A}$ being downward closed means that there is no $\mathbf{u} \in \mathcal{A}$ such that $\mathbf{v} \leq \mathbf{u}$. Equivalently, for each $\mathbf{u} \in \mathcal{A}$, $\mathbf{v} \not\leq \mathbf{u}$. Since $\leq$ is defined pointwise on $\mathbb{B}^m$, this means that for each $\mathbf{u} \in \mathcal{A}$, there exists $i_{\mathbf{u}} \in \{1, \dots, m\}$ such that $\mathbf{v}(i_{\mathbf{u}}) \not\leq \mathbf{u}(i_{\mathbf{u}})$, which can only be the case that $\mathbf{v}(i_{\mathbf{u}}) = 1$ and $\mathbf{u}(i_{\mathbf{u}}) = 0$. Consider the $\neg$-clauses $\gamma := \{x_{i_{\mathbf{u}}} \mid \mathbf{u} \in \mathcal{A}\}$ picked as above. On one hand, each $\mathbf{u} \in \mathcal{A}$ satisfies γ because $\mathbf{u}(x_{i_{\mathbf{u}}}) = 0$, so $\gamma \in \Gamma$. On the other hand, $\mathbf{v}(\gamma) = 1$ because $\mathbf{v}(x_{i_{\mathbf{u}}}) = 1$ for all $\mathbf{u} \in \mathcal{A}$, thus $\mathbf{v}$ does not satisfy γ. But this means σ is a $\neg$-clause in Γ not satisfied by $\mathbf{u}$, contradiction with the assumption that $\mathbf{u} \in \mathit{Sat}(\Gamma)$. □

Finally we point out the connection between sets of ¬-clauses and matrix diagrams in Definition 5.3.2. This is not hard to see given Corollary 5.4.14 and the semantics of matrix diagrams as monotone relations generated by monotone functions.

Definition 5.4.15. Let $\Sigma = \{\sigma_1, \ldots, \sigma_n\}$ be a set of set of ¬-clauses over Var. We define the *matrix diagram generated by* Σ to be $\mathit{mat}_\Sigma \colon m \to n$ such that its entries $\delta_{ij}^{\mathit{mat}_\Sigma}$ is —— if x_i appears in σ_j, and is —●○— otherwise.

The matrix diagrams generated by a set of ¬-clauses satisfies the following semantic property.

Proposition 5.4.16. *Following the notation from Definition 5.4.15, for arbitrary* $\mathbf{b} \in \mathbb{B}^m$*, there exists* $\mathbf{a} \in \mathbb{B}^n$ *such that for each* $i = 1, \ldots, n$*,* $\mathbf{a}(i) = 0$ *and* $(\mathbf{b}, \mathbf{a}) \in [\![\mathit{mat}_\Sigma]\!]$ *if and only if* $\mathbf{b}$ *satisfies* σ_i*.*

Proof. Spelling out the semantics of mat_Σ, $(\mathbf{b}, \mathbf{a}) \in [\![\mathit{mat}_\Sigma]\!]$ means that $\mathbf{b}(\sigma_i) \leq \mathbf{a}(i)$, for all $i = 1, \ldots, n$. Thus if $\mathbf{a}(i) = 0$, then $\mathbf{b}(\sigma_i) \leq \mathbf{a}(i) = 0$ implies $\mathbf{b}(\sigma_i)$ satisfies σ_i. The other direction of the reasoning also holds. □

Intuitively, (—●⊏, ⊐○—) matrix diagram corresponds to the conjunctive reading of ¬-clauses: the (—●⊏, —●) make copies of the variables because each variable may appear several times in a set of ¬-clauses; the (⊐○—, ○—) part perform conjunctions, one for each ¬-clause.

5.4.2 Fullness

Now we use ¬-clauses as a bridge for proving fullness (as well as faithfulness in the next subsection) of $[\![\cdot]\!] : \mathbb{SATA}^{\mathit{func}} \to \mathsf{MonRF}_{\mathbb{B}}$. In a nutshell, starting from a monotone function $f \colon \mathbb{B}^m \to \mathbb{B}$, one construct a set of ¬-clauses Σ_f (Definition 5.4.17), which then induces a $\mathbb{SATA}^{\mathit{func}}$ morphism d_f (Definition 5.4.19). Both steps 'preserve' the semantics (Proposition 5.4.18, Proposition 5.4.21), thus the interpretation of d_f is exactly the monotone relation representing f (*i.e.* $[\![d_f]\!] = \mathcal{R}(f)$).

Definition 5.4.17. Let $f \colon \mathbb{B}^m \to \mathbb{B}$ be a monotone function. Γ_f is defined as the unique minimal set of ¬-clauses over $\mathit{Var} = \{x_1, \ldots, x_m\}$ such that $\mathit{Sat}(\Gamma_f) = f^{-1}(0)$.

Such Γ_f is well-defined because of Corollary 5.4.14 and that $f^{-1}(0)$ is always a downward closed set for a monotone f. Moreover, f can be explicitly expressed (up to extensional equivalence) as the conjunction of ¬-clauses in Γ_f.

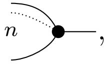
Proposition 5.4.18. *Following the notation from Definition 5.4.17, suppose* $\Gamma_f = \{\sigma_1, \ldots, \sigma_k\}$*, where each* $\sigma_i = \{x_{i_1}, \ldots, x_{i_{\ell_i}}\}$*, then*

$$f = \lambda x_1 \ldots x_m . \bigvee_{i=1}^{k} (x_{i_1} \wedge \cdots \wedge x_{i_{\ell_i}})$$

Proof. Since both functions can take value only in $\mathbb{B}$, the statement amounts to showing that, for arbitrary $\mathbf{b} \in \mathbb{B}^m$, $f(\mathbf{b}) = 0$ if and only if $\bigvee_{i=1}^{k}(b_{i_1} \wedge \cdots \wedge b_{i_{\ell_i}}) = 0$.

Suppose $f(\mathbf{b}) = 0$. Since $\mathit{Sat}(\Gamma_f) = f^{-1}(0)$, this implies $\mathbf{b}$ satisfies Γ_f, in the sense that for each $\sigma_i \in \Gamma_f$, $b_{i_1} \wedge \cdots \wedge b_{i_{\ell_i}} = 0$. So $\bigvee_{i=1}^{k}(b_{i_1} \wedge \cdots \wedge b_{i_{\ell_i}}) = 0$. The inverse of the above reasoning is also true, which means that $\bigvee_{i=1}^{k}(b_{i_1} \wedge \cdots \wedge b_{i_{\ell_i}}) = 0$ implies $f(\mathbf{b}) = 0$. □

Next the monotone function $\lambda x_1 \dots x_m. \bigvee_{i=1}^{k}(x_{i_1} \wedge \cdots \wedge x_{i_{\ell_i}})$ can be represented diagrammatically.

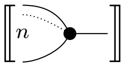
Definition 5.4.19. Let Σ be a set of $\neg$-clauses over $\mathit{Var} = \{x_1, \dots, x_m\}$ of size n. We define d_Σ^{func} as the $\mathbb{SATA}^{func}$-morphism of type $m \to 1$ composed as mat_Σ ; , where mat_Σ is the matrix diagram whose entries $\delta_{ij}^{\mathit{mat}_\Sigma}$ are defined as follows:

$$\delta_{ij}^{\mathit{mat}_\Sigma} = \begin{cases} \text{——} & \text{if } x_i \text{ appears in } \sigma_j \\ \text{—●○—} & \text{otherwise} \end{cases}$$

Lemma 5.4.20. *For arbitrary* $\mathbf{v} \colon \mathit{Var} \to \mathbb{B}$,

$$\mathbf{v} \in \mathit{Sat}(\Sigma) \textit{ if and only if } ((\mathbf{v}(x_1), \dots, \mathbf{v}(x_m)), 0) \in [\![d_\Sigma]\!]$$

Proof. We continue the notation in Definition 5.4.19, and suppose further that Σ is $\{\sigma_1, \dots, \sigma_n\}$ where each σ_i is of the form $\{x_{i_1}, \dots, x_{i_{k_i}}\}$.

$(\mathbf{v}, 0) \in [\![d_\Sigma]\!] \iff \exists \mathbf{u} \in \mathbb{B}^n \colon (\mathbf{v}, \mathbf{u}) \in [\![\mathit{mat}_\Sigma]\!], (\mathbf{u}, 0) \in$ 

$$\begin{aligned} &\iff (\mathbf{v}, 0^n) \in [\![\mathit{mat}_\Sigma]\!] \\ &\iff \forall i \in \{1, \dots, n\} \colon \mathbf{v}(x_{i_1}) \wedge \cdots \wedge \mathbf{v}(x_{i_{k_i}}) \leq 0 \\ &\iff \mathbf{v} \in \mathit{Sat}(\Sigma) \end{aligned}$$

□

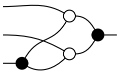
Proposition 5.4.21. $d_{\Gamma_f}^{func}$ *represents* f*, namely* $\left[\!\left[d_{\Gamma_f}^{func}\right]\!\right] = \mathcal{R}(f)$*.*

Proof. Immediate from Proposition 5.4.18 and Lemma 5.4.20.

On one hand, if $(b_1, \dots, b_m, 0) \in \left[\!\left[d_{\Gamma_f}^{func}\right]\!\right]$, then by Lemma 5.4.20, $(b_1, \dots, b_m) \in \mathit{Sat}(\Gamma_f)$. By Proposition 5.4.18, $f(b_1, \dots, b_m) = 0$, thus $(b_1, \dots, b_m, 0) \in \mathcal{R}f$.

On the other hand, given $(b_1, \dots, b_m, 0) \in \mathcal{R}f$, $f(b_1, \dots, b_m) \leq 0$ namely $f(b_1, \dots, b_m) = 0$. By Definition 5.4.17, $(b_1, \dots, b_m) \in \mathit{Sat}(\Gamma_f)$, and Lemma 5.4.20 implies that $(b_1, \dots, b_m, 0) \in \left[\!\left[d_{\Gamma_f}^{func}\right]\!\right]$. □

Example 5.4.22. Let $\mathit{Var} = \{x_1, x_2, x_3\}$, and monotone function $f \colon \mathbb{B}^3 \to \mathbb{B}$ map $111, 011, 101$ to 1, and map other elements in $\mathbb{B}^3$ to 0. In other words, $f(b_1 b_2 b_3) = (b_1 \wedge b_3) \vee (b_2 \wedge b_3)$. The downward closed subset $f^{-1}(0)$ generates the minimal sets of $\neg$-clauses $\Sigma_f = \{x_1 \wedge x_3 =$

$0, x_2 \wedge x_3 = 0\}$. Thus by Definition 5.4.19, Σ_f gives rise to the diagram $d_{\Sigma_f} =$ , whose interpretation $[\![d_{\Sigma_f}]\!]$ is exactly the monotone relation $\mathcal{R}(f)$ representing f:

$$\begin{aligned}((x_1,x_2,x_3),y) \in [\![d_{\Sigma_f}]\!] &\iff \exists u_1,u_2,v_1,v_2 \in \mathbb{B} : x_3 \le u_1,u_2; x_1 \wedge u_1 \le v_1; x_2 \wedge u_2 \le v_2; v_1 \wedge v_2 \le y\\ &\iff \exists v_1,v_2 \in \mathbb{B} : x_1 \wedge x_3 \le v_1; x_2 \wedge x_3 \le v_2; v_1 \wedge v_2 \le y\\ &\iff x_1 \wedge x_3 \le y \;\&\; x_2 \wedge x_3 \le y\\ &\iff (x_1 \wedge x_3) \vee (x_2 \wedge x_3) \le y\\ &\iff f(x_1x_2x_3) \le y\\ &\iff ((x_1,x_2,x_3),y) \in \mathcal{R}(f)\end{aligned}$$

Now we are ready to prove fullness of $[\![\cdot]\!] : \mathbb{SATA}^{func} \to \mathsf{MonRF}_{\mathbb{B}}$.

Proposition 5.4.23 (Fullness). *For every monotone function $f \colon \mathbb{B}^m \to \mathbb{B}$, there exists a Syn^{func}-morphism $d \colon m \to 1$ such that $[\![d]\!] = \mathcal{R}(f)$.*

Proof. The strategy is as follows: starting from f, the pre-image $f^{-1}(0)$ forms a downward-closed subset of $\mathbb{B}^m$, thus the set of satisfying assignments of some set of $\neg$-clauses, and it generates a morphism d with the desired semantics.

Now we fill in the details of every single step. Let Σ_f be the unique minimal set of $\neg$-clauses such that $\mathit{Sat}(\Sigma_f) = f^{-1}(0)$, as in Definition 5.4.17. Then Σ_f determines a diagram $d_{\Sigma_f} \colon m \to 1$ as in Definition 5.4.19, which satisfies that for each valuation $\mathbf{v} \colon \mathit{Var} \to \mathbb{B}$, $\mathbf{v} \in \mathit{Sat}(\Sigma_f)$ if and only if $((\mathbf{v}(x_1),\dots,\mathbf{v}(x_m)),0) \in [\![d_{\Sigma_f}]\!]$. We show that $[\![d_{\Sigma_f}]\!] = \mathcal{R}(f)$. For arbitrary $(\mathbf{b},a) \in \mathbb{B}^m \times \mathbb{B}$, if $a = 1$, then $(\mathbf{b},1)$ is always in both $[\![d_{\Sigma_f}]\!]$ and $\mathcal{R}(f)$. So it suffices to consider the case of $a = 0$:

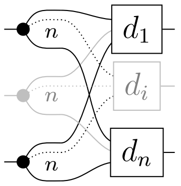
$$\begin{aligned}(\mathbf{b},0) \in [\![d_{\Sigma_f}]\!] &\iff \mathbf{b} \in \mathit{Sat}(\Sigma_f)\\ &\iff \mathbf{b} \in f^{-1}(0)\\ &\iff f(\mathbf{b}) = 0\\ &\iff (\mathbf{b},0) \in \mathcal{R}(f)\end{aligned}$$

Therefore this d_{Σ_f} is a morphism d that satisfies the desired property. □

While the fullness for arbitrary monotone functions $f \colon \mathbb{B}^m \to \mathbb{B}^n$ follows immediately from Proposition 5.4.23, it is helpful to make explicit the morphism d whose interpretation is the given f. In Proposition 5.4.24, we fix a monotone $f \colon \mathbb{B}^m \to \mathbb{B}^n$. We know that $f = (f_1,\dots,f_n)$ where each $f_i \colon \mathbb{B}^m \to \mathbb{B}$ is monotone. The idea is simple: we make m copies of the input and send each copy to one of those d_i.

Proposition 5.4.24. *Let d_i be $d_{\Sigma_{f_i}}$, then the diagram $d \colon m \to n$ defined as below satisfies*

$[\![d]\!] = \mathcal{R}(f)$:



(5.18)

Proof. According to Proposition 5.4.23, each d_i satisfies that $[\![d_i]\!] = \mathcal{R}(f_i)$, then for arbitrary $\mathbf{b} \in \mathbb{B}^m$ and $\mathbf{a} \in \mathbb{B}^n$,

$$\begin{aligned}
(\mathbf{b}, \mathbf{a}) \in [\![d]\!] &\iff \forall i \in \{1, \ldots, m\}, \exists \mathbf{c}^i \in \mathbb{B}^m \colon \mathbf{b} \leq \mathbf{c}^i, (\mathbf{c}^i, \mathbf{a}) \in [\![d_i]\!] \\
&\iff \forall i \in \{1, \ldots, m\}, \exists \mathbf{c}^i \in \mathbb{B}^m \colon \mathbf{b} \leq \mathbf{c}^i, (\mathbf{c}^i, a_i) \in \mathcal{R}(f_i) \\
&\iff \forall i \in \{1, \ldots, m\}, \exists \mathbf{c}^i \in \mathbb{B}^m \colon \mathbf{b} \leq \mathbf{c}^i, f_i(\mathbf{c}^i) \leq a_i \\
&\iff \forall i \in \{1, \ldots, m\} \colon f_i(\mathbf{b}) \leq a_i \\
&\iff f(\mathbf{b}) \leq \mathbf{a} \\
&\iff (\mathbf{b}, \mathbf{a}) \in \mathcal{R}(f)
\end{aligned}$$

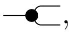
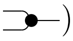
This means $[\![d]\!] = \mathcal{R}(d)$. □

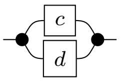
5.4.3 Faithfulness

While the proof of fullness only uses limited properties of system *SATA*$^{\mathit{func}}$ (in fact only the property of matrix diagrams), for faithfulness we need the full power of *SATA*$^{\mathit{func}}$ to show that every inequation in $\mathsf{MonRF}_{\mathbb{B}}$ is deducible in the inequational theory *SATA*$^{\mathit{func}}$. The central result of this subsection is stated as follows.

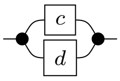
Proposition 5.4.25 (Faithfulness)**.** *For arbitrary* $\mathsf{Syn}^{\mathit{func}}$*-morphism* $c, d \colon m \to n$*, if* $[\![c]\!] \subseteq [\![d]\!]$*, then* $c \leq_{SATA^{\mathit{func}}} d$*.*

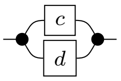
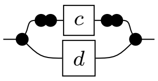
Equivalently, faithfulness means that, if $[\![c]\!] = \mathcal{R}(f)$ and $[\![d]\!] = \mathcal{R}(g)$ for some monotone functions $f, g \colon \mathbb{B}^m \to \mathbb{B}^n$ such that $f \geq g$, then $c \leq_{SATA^{\mathit{func}}} d$. We adopt a so-called *normal form proof* strategy, which is prevalent in many work on diagrammatic calculus. The proof consists of two steps: first we show that every $\mathbb{SATA}^{\mathit{func}}$-morphism has a *canonical form* in which generators appear in a certain order; second we prove that from every semantic object (monotone functions in this case) we can 'read off' a diagrammatic representative called *normal form*, which can be easily derivable from canonical form diagrams.

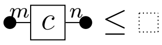
Before entering the normal form proof, we observe that we can reduce the above faithfulness involving inequalities (Proposition 5.4.25) into one that only involves equalities (Proposition 5.4.35), using the following lemma that links an inequality with an equality using the (, ) structure in our language.

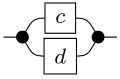
Lemma 5.4.26. *For arbitrary* $\mathbb{SATA}^{\mathit{func}}$*-morphisms* $c, d \colon m \to n$*,* 

= c implies $c \leq d$*.*

Proof. Suppose  $= c$, then

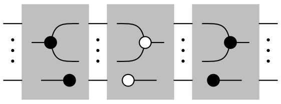
$c =$  $\leq$  $\leq d$

The last step uses the fact that , provable via a simple induction on the structure of diagram c. □

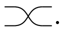
Intuitively this amounts to saying that in a lattice, $x \wedge y \iff x \wedge y = x$. The other direction of this statement is also true, namely we have  $= c$ if and only if $c \leq d$. For our proof of faithfulness however, only one direction (the easier one) is necessary.

Now we come to the normal form proof. As suggested by the form of the diagrams for the diagram generated by sets of ¬-clauses d_Σ from Definition 5.4.19, we propose the following canonical form to specify the order of generators in diagrams.

Definition 5.4.27. A $\mathbb{SATA}^{func}$-morphism $d\colon m \to n$ is of *canonical form* if it has the following structure:



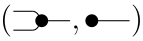
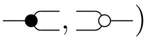
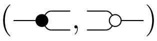
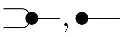
(5.19)

Each grey block represents a diagram that is formed solely of the generators in the block, possibly including some permutation of diagrams via .

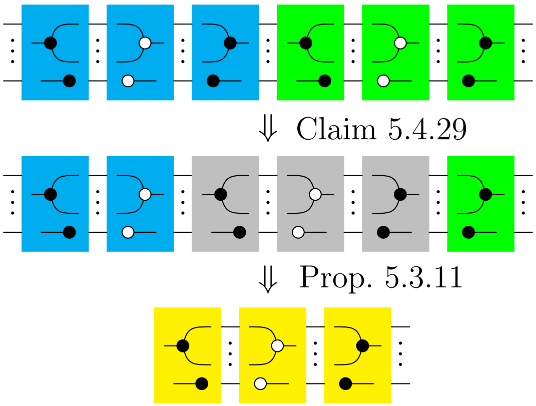
Lemma 5.4.28. *Every* $\mathbb{SATA}^{func}$*-morphism* $d\colon m \to n$ *is equal to one of canonical form.*

Proof. We prove by induction on the free construction of Syn^{func} diagrams. For the induction basis, every generator already exists in the canonical form (5.19).

For the induction step, it suffices to show that canonical form diagrams are closed under both parallel and sequential compositions, up to equivalence in $SATA^{func}$. The case for parallel composition is immediate. As for sequential composition, it suffices to prove the following claim:

Claim 5.4.29. If c_1 is a () diagram and c_2 is a () matrix diagram, then there exist a () matrix diagram d_1 and a () diagram such that $c_1 \mathbin{;} c_2 = d_1 \mathbin{;} d_2$.

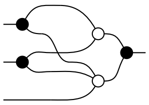
Given this claim, the original statement follows via the following reasoning:



In words, given two canonical form diagrams (colored in blue and green, respectively), we first apply Claim 5.4.29 to change the order of the last blue block and the first two blue block and get the grey blocks. Then matrix completeness says the concatenation of two matrix diagrams (*i.e.* the first four blocks) is equal to a matrix diagram (*i.e.* the first two yellow blocks). The result is a diagram in canonical normal form. The proof of Claim 5.4.29 adopts an induction proof, relying essentially on axioms (A9)-(A12) and (dst1). □

Now, given a monotone function $f\colon \mathbb{B}^m \to \mathbb{B}^n$, from Proposition 5.4.24 we know that there exists a $\mathbb{SATA}^{func}$-morphism $d\colon m \to n$ such that $[\![d]\!] = \mathcal{R}(f)$. Such diagrams as created in Definition 5.4.19 are always of canonical form by construction: each of them is formed of a (—●⊂, ⊃○—) matrix diagram followed by multiple ⊃●—. However, similar to that two different sets of ¬-clauses may have the same set of solutions, two such canonical form diagrams may look different but have the same interpretation. To pick out a representative diagram for each semantic object, we introduce normal form diagrams. Intuitively normal form diagrams correspond to the minimal sets of ¬-clauses. Before giving the formal definition, it is helpful to see an example of a canonical diagram which is not of normal form.

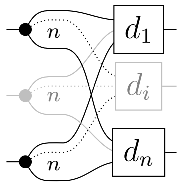
Example 5.4.30. We consider the following diagram $d_1\colon 3 \to 1$:



(5.20)

One can read off the function $f_1\colon \mathbb{B}^3 \to \mathbb{B}$ from d_1 as $f_1 \equiv \lambda x_1 x_2 x_3.(x_1 \wedge x_2) \vee (x_1 \wedge x_2 \wedge x_3)$, in the sense that $[\![d_1]\!] = \mathcal{R}(f_1)$. Observe that in the expression of f_1, the ¬-clause $x_1 \wedge x_2 \wedge x_3$ (*i.e.* $\{x_1, x_2, x_3\}$) is redundant given the ¬-clause $x_1 \wedge x_2$.

Definition 5.4.31. A canonical form diagram $d\colon m \to 1$ is of *normal form* if, suppose A is the representing $k \times m$-matrix of the (—●⊂, ⊃○—)-matrix part of d, then there are no distinct rows i, j in A such that $A_{i\ell} \leq A_{j\ell}$ for all $\ell = 1, \ldots, k$. A canonical form diagram $d\colon m \to n$ is of *normal form* if d is composed as follows where each $d_i\colon m \to 1$ is of normal form

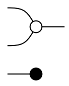


(5.21)

Using the intuition above for normal form diagrams and minimal sets of $\neg$-clauses, we can equivalently say that a canonical diagram $d\colon m \to 1$ is of normal form if $d = d_\Sigma$ for some minimal set of $\neg$-clauses Σ over $\{x_1, \ldots, x_m\}$.

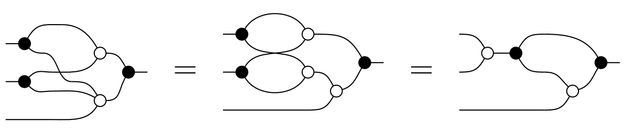
Remark 5.4.32. One may note that the diagram (5.21) in defining normal form diagrams with codomains ≥ 1 is exactly the same as diagram (5.18). This should not be surprising: both use the same mechanism to connect multiple diagrams of type $m \to 1$ into a single diagram of type $m \to n$ while maintaining certain properties.

Example 5.4.33. We continue Example 5.4.30. The representing matrix A_1 of diagram d_1 in (5.20) is $A_1 = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 1 & 1 \end{pmatrix}$, which fails the condition in Definition 5.4.31 because $A_{1\ell} \leq A_{2\ell}$ for each $\ell \in \{1, 2, 3\}$. The following diagram d_2 is of normal form:



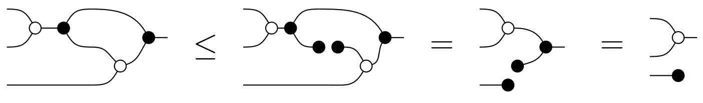
(5.22)

and it has the same semantics as d_1, namely $[\![d_1]\!] = [\![d_2]\!]$. Looking at their corresponding functions, d_2 represents the function $f_2 \equiv \lambda x_1 x_2 x_3 . x_1 \wedge x_2$, which basically drops the redundant $\neg$-clause in f_1 (see Example 5.4.30). Moreover, we can prove the equivalence of d_1 and d_2 in *SATA*$^{\mathit{func}}$. We massage d_1 to get

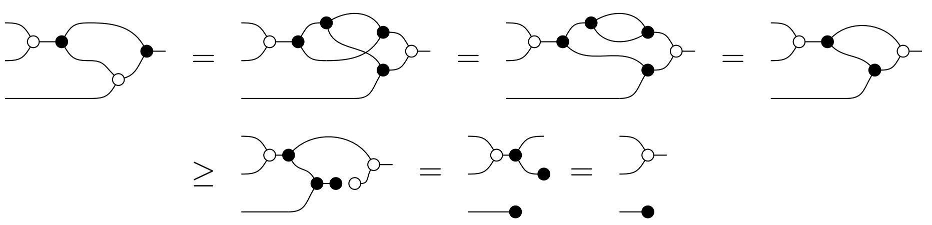


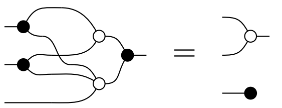
(5.23)

Then we prove the above diagram equals d_2 by showing the two inequalities separately. On one hand,



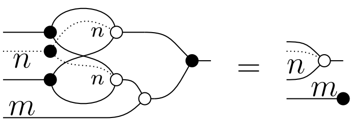
On the other hand,



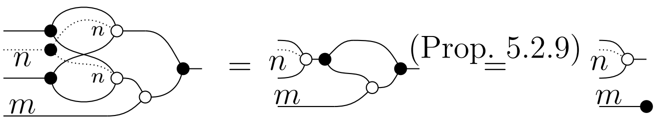
So we can conclude that  .

As hinted by the previous example, we have a diagrammatic counterpart of dropping redundant ¬-clauses.

Lemma 5.4.34. *For arbitrary natural numbers m, n,*



Proof.

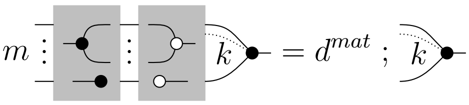


□

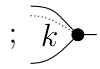
Semantically, the above lemma says $(x_1 \wedge \cdots \wedge x_n) \vee (x_1 \wedge \cdots \wedge x_n \wedge u_1 \wedge \cdots \wedge u_m) = x_1 \wedge \cdots \wedge x_n$. Then we show the faithfulness that involves only equalities (but not inequalities in general).

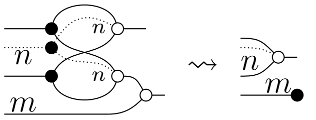
Proposition 5.4.35. *For arbitrary $\mathbb{SATA}^{func}$-morphisms $c, d\colon m \to n$, if $[\![c]\!] = [\![d]\!]$, then $c =_{SATA^{func}} d$.*

Proof. We start with an arbitrary $d\colon m \to 1$, and show that it is equivalent to a normal form diagram $e\colon m \to 1$. Denote the monotone function represented by $[\![d]\!]$ as $f\colon \mathbb{B}^m \to \mathbb{B}$. First, by Lemma 5.4.28, without loss of generality we can assume that d is of canonical form already, say as follows (so d^{mat} is the matrix diagram part):



Then the matrix diagram $d^{mat}\colon m \to k$ represents a $k \times m$ Boolean matrix, say D (see Definition **??**).

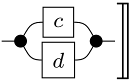
We now massage D into a matrix that satisfies the condition for normal form diagram in Definition 5.4.31. Suppose D is does not satisfy that condition, namely there are two distinct rows i, i' in D such that $D_{i\ell} \leq D_{i'\ell}$ for all $\ell = 1, \ldots, k$, then we let A' be the same matrix as D except for the i' row: $D_{i'\ell} = 0$ for all $\ell = 1, \ldots, k$. Then we continue to check if D' satisfies that condition of matrix in Definition 5.4.31. Since D has finitely many rows, this procedure terminates in finitely many steps, and the resulting matrix, which we denote by $\hat{D}$, satisfies that there are no two distinct rows i and i' such that $\hat{D}_{i\ell} \leq \hat{D}_{i'\ell}$ for all $i = 1, \ldots, k$. Let $\hat{d}^{mat}$ be the (—●⊂, ⊃○—) matrix diagram for $\hat{D}$ (see Proposition 5.3.11), and $\hat{d} = \hat{d}^{mat}$; . By its construction, $\hat{d}$ is of normal form, and we claim that $\left[\!\!\left[\hat{d}\right]\!\!\right] = [\![d]\!]$ and $\hat{d} =_{SATA^{func}} d$. By the soundness, it suffices to prove the syntactic equation. By the correspondence between Boolean matrices and (—●⊂, ⊃○—) matrix diagrams, in the procedure above to turn D into $\hat{D}$, each step that rewrites a row i' satisfying that there is some other row i with $D_{i\ell} \leq D_{i'\ell}$ for all $\ell = 1, \ldots, k$ into a row of 0's corresponds to the following transformation diagrammatically:

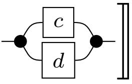


While the resulting matrix diagrams are not equal, Lemma 5.4.34 says that the resulting canonical diagrams are equal, therefore $d =_{SATA^{func}} \hat{d}$.

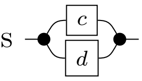
Now given canonical diagrams $c, d \colon m \to 1$ satisfying $[\![c]\!] = [\![d]\!]$, and assume that their matrix diagrams correspond to Boolean matrices C and D, respectively. Then applying the aforementioned transformation to C and D respectively will result in two matrices $\hat{C}$ and $\hat{D}$. The entries of these matrices correspond to the least sets of $\neg$-clauses whose solution sets are $\{\mathbf{a} \in \mathbb{B}^m \mid (\mathbf{a}, 0) \in [\![c]\!]\}$ and $\{\mathbf{b} \in \mathbb{B}^m \mid (\mathbf{a}, 0) \in [\![d]\!]\}$, respectively. But since $[\![c]\!] = [\![d]\!]$, this means that $\hat{C}$ and $\hat{D}$ are equivalent up to permutations of their rows. This means that their corresponding matrix diagrams, $\hat{c}$ and $\hat{d}$, are equal. Therefore $c = \hat{c} = \hat{d} = d$. □

Finally the faithfulness for inequalities follows from the following simple line of reasoning, using all the results we have established so far.

Proof of Proposition 5.4.25. Let $c, d \colon m \to n$ be two $\mathbb{SATA}^{func}$-morphisms such that $[\![c]\!] \subseteq [\![d]\!]$. Then we can calculate $[\![$  $]\!] = [\![c]\!]$ as follows. For arbitrary $\mathbf{a} \in \mathbb{B}^m$ and $\mathbf{b} \in \mathbb{B}^n$,

$(\mathbf{a}, \mathbf{b}) \in [\![$  $]\!]$

$$\begin{aligned} &\iff \exists \mathbf{c}_1, \mathbf{c}_2 \in \mathbb{B}^m, \exists \mathbf{d}_1, \mathbf{d}_2 \in \mathbb{B}^n \colon (\mathbf{a}, (\mathbf{c}_1, \mathbf{c}_2) \in [\![\text{—●⊂}]\!]), ((\mathbf{d}_1, \mathbf{d}_2), \mathbf{b}) \in [\![\text{⊃●—}]\!], \\ &\qquad\qquad (\mathbf{c}_1, \mathbf{d}_1) \in [\![c]\!], (\mathbf{c}_2, \mathbf{d}_2) \in [\![d]\!] \\ &\iff \exists \mathbf{c}_1, \mathbf{c}_2 \in \mathbb{B}^m, \exists \mathbf{d}_1, \mathbf{d}_2 \in \mathbb{B}^n \colon \mathbf{a} \leq \mathbf{c}_1, \mathbf{a} \leq \mathbf{c}_2, \mathbf{d}_1 \leq \mathbf{b}, \mathbf{d}_2 \leq \mathbf{b}, \\ &\qquad\qquad (\mathbf{c}_1, \mathbf{d}_1) \in [\![c]\!], (\mathbf{c}_2, \mathbf{d}_2) \in [\![d]\!] \\ &\iff (\mathbf{a}, \mathbf{b}) \in [\![c]\!], (\mathbf{a}, \mathbf{b}) \in [\![c]\!] \\ &\iff (\mathbf{a}, \mathbf{b}) \in [\![c]\!] \end{aligned}$$

By Proposition 5.4.35, this entails  $= c$, therefore $c \leq d$ by Lemma 5.4.26. □

5.4.4 $\mathsf{MonFun}_{\mathbb{B}}$ as a traced monoidal category

As a detour, we mention that $\mathsf{MonRF}_{\mathbb{B}}$ itself is already enough to interpret feedback, by showing that $\mathsf{MonRF}_{\mathbb{B}}$ is a traced monoidal category (see Definition 2.3.7) whose trace captures the least fixed point construction.

Definition 5.4.36. For an arbitrary $\mathsf{MonRF}_{\mathbb{B}}$-morphism $R\colon X \times U \to Y \times U$, we define $Tr^U_{X,Y}(R)\colon X \to Y$ to be

$$Tr^U_{X,Y}(R) = \{(x, y) \in X \times Y \mid \exists u, u' \in U \colon u' \leq u \;\&\; ((x, u), (y, u')) \in R\}$$

We need to verify Definition 5.4.36 is legitimate in the sense that $Tr^U_{X,Y}(R)$ is also a monotone relation, given that R is monotone. So for arbitrary $(x, y) \in Tr^U_{X,Y}(R)$, $x' \leq x$ and $y \leq y'$, by definition there exists $u, u' \in U$ such that $u' \leq u$ and $((x, u), (y, u')) \in R$. Since $(x', u) \leq (x, u)$ and $(y, u') \leq (y', u')$, R being a monotone relation implies that $((x', u), (y', u'))$ is also in R, thus $(x', y') \in Tr^U_{X,Y}(R)$.

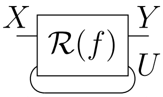
Proposition 5.4.37. *The category* $\mathsf{MonFunc}(\mathbb{B})$ *is traced.*

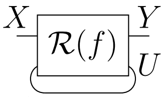
Proof. While we provided a direct proof in Proposition 5.4.37, here we show a simpler proof relying on the fact that $\mathsf{MonRel}_{\mathbb{B}}$ is compact closed, thus having a canonical traced structure. We show that there is a faithful functor $\mathcal{R}\colon \mathsf{MonFunc}(\mathbb{B}) \to \mathsf{MonRel}_{\mathbb{B}}$, which moreover maps the family of functions Tr in $\mathsf{MonFunc}(\mathbb{B})$ (defined below) to the canonical traced structure in $\mathsf{MonRel}_{\mathbb{B}}$. The axioms of traced monoidal categories hold for $\mathsf{MonFunc}(\mathbb{B})$ because their images hold in $\mathsf{MonRel}_{\mathbb{B}}$, and that $\mathcal{R}$ is faithful.

Recall that the candidate of traces in $\mathsf{MonFunc}(\mathbb{B})$ is defined for arbitrary $f\colon X \times U \to Y \times U$ as

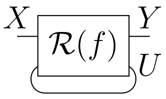
$$Tr^U_{X,Y}(x) = (\pi_1 \circ f)(x, u_x) \tag{5.24}$$

where u_x is the least fixed point of $\lambda u.(\pi_2 \circ f)(x, u)$. In other words, u_x is the smallest $u \in U$ satisfying $f(x, u) = (y, u)$ for some $y \in Y$, and its existence is guaranteed by the monotonicity of f on the complete lattices $\mathbb{B}^n$ ($n \in \mathbb{N}$).

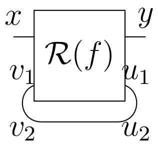
Then we show that $\mathcal{R}(Tr^U_{X,Y}(f)) =$ 

, where 

is the diagrammatic presentation of the morphism $(id_X \otimes cup_U) \,;\, \mathcal{R}f \,;\, (id_Y \otimes cap_U)$ in $\mathsf{MonRel}_{\mathbb{B}}$.

$(x, y) \in$ 

$\iff \exists u_1, u_2, v_1, v_2 \in U : 1 \leq v_1 \wedge v_2, v_2 \leq u_2, u_1 \wedge u_2 \leq 0, ((x, v_1), (y, u_1)) \in \mathcal{R}(f)$

(See 

for an illustration.)

$\iff \exists u_1, v_1 \in U : u_1 \leq v_1 \;\&\; ((x, v_1), (y, u_1)) \in \mathcal{R}(f)$

$\iff \exists u, v \in U : u \leq v \;\&\; f(x, v) \leq (y, u)$

$\iff \exists v \in U : f(x, v) \leq (y, v)$

$\iff \exists v \in U : f(x, \mu u.\pi_2 \circ f(x, u)) \leq f(x, v) \leq (y, v)$

$\iff Tr^U_{X,Y}(f)(x) \leq y$

$$\iff (x,y) \in \mathcal{R}(\mathit{Tr}^U_{X,Y}(f))$$

Since the traced proposed for $\mathsf{MonFun}_{\mathbb{B}}$ are indeed represented by the canonical traces in $\mathsf{MonRel}_{\mathbb{B}}$ via a faithful functor, we can conclude that $\mathsf{MonRel}_{\mathbb{B}}$ is also traced. □

One open question is a sound and complete axiomatisation of $\mathsf{MonFun}_{\mathbb{B}}$ as a traced monoidal category. Such an axiomatisation is sufficient for some application of *SATA*, for instance the equivalence of definite propositional logic programs.

5.5 Soundness and Completeness of monotone Boolean relations

Now we turn to our main subject of interest, monotone relations over finite Boolean algebras. In particular, we devote this section to the following isomorphism between two poset-enriched categories.

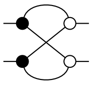
Theorem 5.5.1. $\mathbb{SATA} \cong \mathsf{MonRel}_{\mathbb{B}}$

Spelling it out, Theorem 5.5.1 amounts to the following theorem, stated in a more logical fashion.

Theorem 5.5.2. *For arbitrary* $\mathbb{SATA}$*-morphisms* c *and* d*,* $c \leq_{\mathbb{SATA}} d$ *if and only if* $\llbracket c \rrbracket \subseteq \llbracket d \rrbracket$*.*

In other words, we show that the equational theory *SATA* is a complete axiomatisation of monotone relations over (finite) Boolean algebras.

The 'only if' direction of Theorem 5.5.21 is the soundness part. It is straightforward as we simply need to show that each axiom is a valid semantic (in)equality. We take (C2) as an example, and omit the rest which are straightforward.

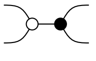
$\llbracket$  $\rrbracket$

$$=\{((x_1,x_2),(y_1,y_2)) \in \mathbb{B}^4 \mid \exists z_1,z_2,z_3,z_4 \in \mathbb{B} : x_1 \leq z_1 \wedge z_2, x_2 \leq z_3 \wedge z_4, z_1 \wedge z_3 \leq y_1, z_2 \wedge z_4 \leq y_2\}$$

$$=\{((x_1,x_2),(y_1,y_2)) \in \mathbb{B}^4 \mid \exists z_3,z_4 \in \mathbb{B} : x_2 \leq z_3 \wedge z_4, x_1 \wedge z_3 \leq y_1, x_1 \wedge z_4 \leq y_2\}$$

$$=\{((x_1,x_2),(y_1,y_2)) \in \mathbb{B}^4 \mid \exists z_3,z_4 \in \mathbb{B} : x_1 \wedge x_2 \leq y_1, x_1 \wedge x_2 \leq y_2\}$$

$$=\{((x_1,x_2),(y_1,y_2)) \in \mathbb{B}^4 \mid x_1 \wedge x_2 \leq y_1 \wedge y_2\}$$

$= \llbracket$  $\rrbracket$

We now focus on the 'if' direction, namely the completeness. Before the technical developments, we provide a roadmap of the proof. We first show that we can make two simplifying assumptions: (1) the completeness statement about inequalities/inclusions can be reduced to one that involves only equalities (Lemma 5.5.3); (2) We simplify the problem further by showing that we can restrict the proof of completeness to diagrams of the type $m \to 0$ without loss of generality (Lemma 5.5.4).

To prove completeness for $m \to 0$ diagrams, we again adopt a normal form argument, similar to that used in Section 5.4.3. Yet of course we need a different normal form this time. The structure of the normal form argument for $\mathbb{SATA}$ is as follows.

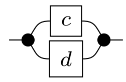
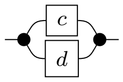
- First, we show that every diagram is equal in *SATA* to one in *traced canonical form* (Lemma 5.5.5), a form that restricts the order in which generators can appear, yet allowing certain loops to connect wires on the right to wires on the left.

- Next, we give a procedure to rewrite any traced canonical form diagram into one in *pre-normal form*, using equations of *SATA* (Lemma 5.5.8). Intuitively, pre-normal form diagrams represent a set of $\neg$-clauses.

- Each downward closed subset is the set of satisfying assignment for a minimal set of $\neg$-clauses (Lemma 5.5.15). The diagrammatic counterpart of this minimal set of clauses is our chosen normal form for each equivalence class of diagrams (Definition 5.5.13). We show that every diagram in pre-normal form is equal to one in normal form (Lemma 5.5.19).

- Finally, since every semantic object has a canonical diagrammatic representative – a normal form diagram – completeness follows from the fact that two diagrams having the same semantics have the same normal form (Proposition 5.5.15).

As explained above, we first show the two results that simplify the completeness reasoning. The completeness for equalities is enough to derive that for inequalities.

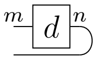
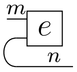
Lemma 5.5.3. *For arbitrary $\mathbb{SATA}$-morphisms c, d, (i) if*  $= c$*, then* $c \leq d$ *and (ii) if* $[\![c]\!] \subseteq [\![d]\!]$ *then* $\Bigg[\!\!\Bigg[$  $\Bigg]\!\!\Bigg] = [\![c]\!]$.

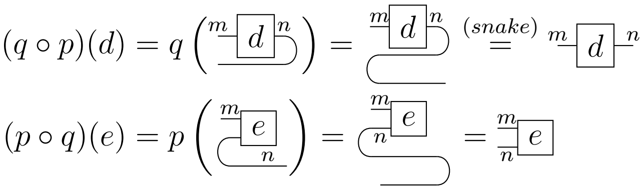
Proof. The proof is similar to that for $\mathbb{SATA}^{func}$-morphisms (Lemma 5.4.26). In particular, neither (i) nor (ii) use specific property of the extra component ⊐∘—. □

In addition, we can restrict the completeness proof to diagrams with codomain 0. This is the consequence of axioms (B9)-(B10) which endow {—∘⊏, —∘, ⊐∘—, ∘—} with the structure of a Frobenius algebra and $\mathbb{SATA}$ with the structure of a compact closed category, allowing us to move wires from the left to the right and vice-versa.

Lemma 5.5.4. *For arbitrary natural numbers m, n,* $\mathsf{Syn}[m, n] \cong \mathsf{Syn}[m + n, 0]$.

Proof. Towards the isomorphism, we define two functions $p \colon \mathsf{Syn}[m, n] \to \mathsf{Syn}[m + n, 0]$ and $q \colon \mathsf{Syn}[m + n, 0] \to \mathsf{Syn}[m, n]$, and prove that they are inverses to each other.

Given arbitrary $d\colon m \to n$ and $e\colon m+n \to 0$, we define $p(d)$ to be , and $q(e)$ to be . Then one can show that $q \circ p$ and $p \circ q$ are identity functions, using the compact closed structure:



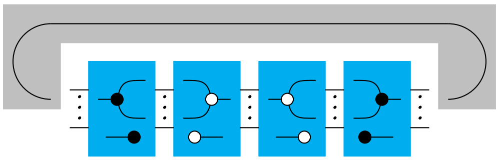
This shows that p and q witness the bijection between $\mathsf{Syn}[m,n]$ and $\mathsf{Syn}[m+n,0]$. □

In view of Lemma 5.5.4, we can simplify the completeness as follows. Given two arbitrary diagram $c, d\colon m \to n$, we can bend their wires on the right port and get two diagrams c', d' of type $m+n \to 0$. Then $c = d$ if and only if $c' = d'$, thus completeness between arbitrary diagrams boils down to on between diagrams of type $k \to 0$ for arbitrary $k \in \mathbb{N}$.

5.5.1 Traced canonical form

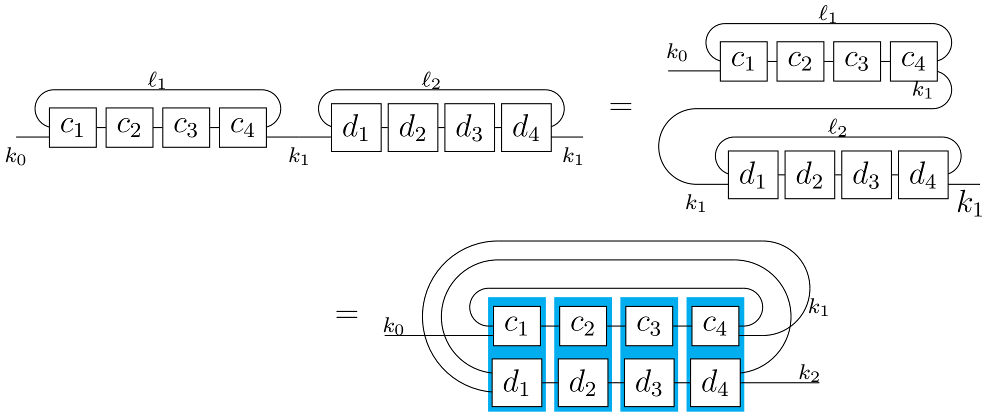
In order to rewrite diagrams $m \to 0$ into pre-normal form, we introduce first the notion of traced canonical form for arbitrary diagrams $m \to n$, which are then bent into diagrams with codomains 0.

Lemma 5.5.5. *Every* Syn*-morphism* $d\colon m \to n$ *has the following* traced canonical form*:*



(5.25)

Proof. We prove by induction on the structure of d. All the base cases (for generators) are obvious because every generator appears in (5.25). For the induction step we need to show that if c and d are two morphisms in traced canonical form, then their parallel and sequential compositions are also of traced-canonical form. For parallel composition this is obvious, and we focus on the case of sequential composition. Suppose the types $c\colon k_0 \to k_1$ and $d\colon k_1 \to k_2$, then $c\,;d$ is:

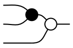
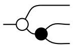


(5.26)

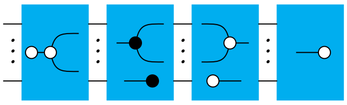
thus again of traced canonical form. □

By bending the right wires of a diagram in traced canonical form to the left, with the mapping $\mathsf{Syn}[m,n] \to \mathsf{Syn}[0,n+m]$ provided by Lemma 5.5.4, any diagram $n \to 0$ is of the following *existential canonical form*. We will need the following notation of oriented boxes, which is important to keep track of when 'sliding' along the traces (see [Sel10]).

Definition 5.5.6 (Oriented box)**.** The *oriented box* notation encode the rotation of diagram $d\colon m \to n$. In particular, denotes the diagram of type $n \to m$ resulted from rotating d by 180°.

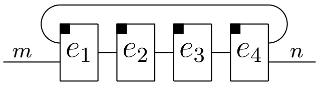
Sometimes is also referred to as the *transpose* of [CK18]. For instance, let $d =$ , then $=$ .

Proposition 5.5.7. *Every diagram of type $k \to 0$ is equal to one in the following* existential canonical form*:*



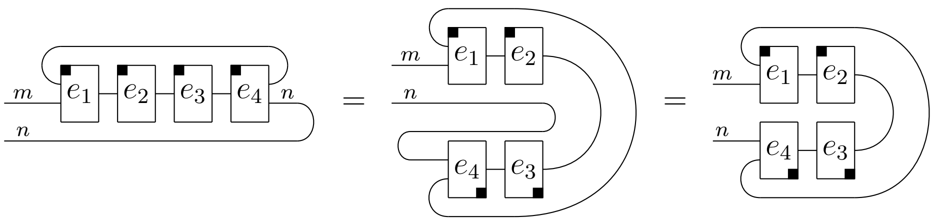
(5.27)

Proof. We pick arbitrary natural numbers m, n satisfying $k = m+n$. Given arbitrary morphism $d\colon k \to 0$, we apply function $q\colon \mathsf{Syn}[m+n,0] \to \mathsf{Syn}[m,n]$ from Lemma 5.5.4, and get $q(d)\colon m \to n$. By Lemma 5.5.5, $q(d)$ is of traced canonical form, say as follows, where each e_i is freely generated by the generators in the i-th blue box in (5.25).



(5.28)

Then we can retrieve d by bending the wires on the right port again, namely as $d = p(q(d))$, which is equivalent to

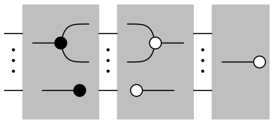


and the last diagram is of existential canonical form (5.27). □

The name *existential canonical form* comes from a resemblance between this form and quantified boolean formulas in prenex form, with the ○◁ behaving as the existential quantifiers in this analogy. In the next part, we show how to remove the ○◁ and obtain pre-normal form diagrams. That proof can be thought of as the diagrammatic counterpart of a quantifier elimination procedure.

5.5.2 Pre-normal form

In this subsection, we show that every diagram is equivalent to one of *pre-normal form*, that is, can be factorised as below:

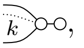


(5.29)

Notice that existential canonical form diagrams (5.27) are close to being in pre-normal form: they are pre-normal form diagrams possibly pre-composed with multiple ○◁. Thus, to prove that every diagram is equal to one in pre-normal form, it suffices to develop a method to eliminate these ○◁ from existential canonical form diagrams, while keeping the pre-normal form structure unchanged. We then explore their close connection with certain Boolean clauses, a key tool in completeness proof. The equational theory allows us to rewrite any diagram to pre-normal form.

Lemma 5.5.8 (Pre-normal form)**.** *Every diagram $m \to 0$ is equal to one of pre-normal form.*

We first have a closer look at pre-normal form diagrams. The first two blocks in (5.29) are matrix diagrams, and they totally determine the pre-normal form diagrams because the last block only contains suitably many —○ so that the resulting diagram has codomain 0. So we refer to the representing matrix of the matrix diagram part of a pre-normal form diagram d simply as the representing matrix of d. Also, Proposition 5.3.11 allows us to identify any two diagrams formed of —●<, —●, ○—, >○— that represent the same matrix. Thus, given a Boolean matrix A, we can define a unique (up to equality in *SATA*) pre-normal form diagram with representing matrix A, the result of postcomposing suitably many —○ to the unique matrix diagram (up to equality in *SATA*) that represents matrix A.

Again, in reasoning about pre-normal forms we will use sets of $\neg$-clauses. We fix a finite set of variables $Var = \{x_1, \ldots, x_m\}$. Recall that a $\neg$-clause is a finite set of variables $\{x_{i_1}, \cdots, x_{i_k}\}$, whose satisfying assignments are all the Boolean valuations $\mathbf{v}$ such that $\mathbf{v}(x_{i_1}) \wedge \cdots \wedge \cdots \mathbf{v}(x_{i_k}) = 0$. To the purpose of the current subsection, such a $\neg$-clause is intended to represent the diagram , as the semantics given by $\{x_{i_1}, \cdots, x_{i_k} \mid x_{i_1} \wedge \cdots \wedge x_{i_k} \leq 0\}$ is exactly the set of satisfying assignments of this set of $\neg$-clauses.

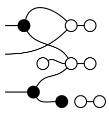
We explain the connection between pre-normal form diagrams and sets of $\neg$-clauses below. To summarise, there is a one-one correspondence up to certain equality.

From pre-normal form to $\neg$-clauses. Given some pre-normal form diagram $d \colon m \to 0$, we define clause_d to be the set of $\neg$-clauses over m variables (say $x_1, \ldots, x_m$) as follows. Each row r of the representing matrix A of d generates a $\neg$-clause φ such that $x_i \in \varphi$ if and only if the i-th element $r[i] = 1$, and clause_d is the set of all $\neg$-clauses generated by the rows of A. The diagram d and the set of $\neg$-clauses clause_d are semantically equivalent in the following sense.

Proposition 5.5.9. *Let $d \colon m \to 0$ be a pre-normal form diagram. Then $[\![d]\!]$ is exactly the set of all satisfying assignments of* clause_d*.*

Proof. Suppose A is the representing matrix of (the matrix diagram part of) d. Note that $[\![d]\!]$ is exactly those $(x_1, \ldots, x_m)$ such that $A\begin{pmatrix} x_1 \\ \vdots \\ x_m \end{pmatrix} \leq 0$. But spelling out the matrix multiplication is exactly the set of $\neg$-clauses clause_d. □

Example 5.5.10. Consider the following pre-normal form diagram d. It generates the set of $\neg$-clauses $\mathsf{clause}_d = \{\{x_1, x_2\}, \{x_1, x_3\}, \{\}\}$. The semantics of d is $[\![d]\!] = \varnothing$, which is exactly the set of all satisfying assignments of $\mathsf{clause}d$.

$d =$ 

From $\neg$-clauses to pre-normal form. Conversely, given a set of $\neg$-clauses Φ over Var, we can define a diagram $\mathsf{diag}_\Phi \colon m \to 0$ in pre-normal form. Assume that $|\Phi| = n$, and we list the $\neg$-clauses in Φ using lexicographical order as $\varphi_1, \ldots, \varphi_n$ (for instance, $\{x_1, x_3\}$ appears before $\{x_2\}$). Then diag_Φ is defined as the pre-normal form diagram whose representing Boolean matrix A (of size $n \times m$) is given by $A_{ij} = 1$ precisely if $x_j \in \varphi_i$. Again, Φ and diag_Φ are semantically equivalent in the following sense.

Lemma 5.5.11. *For Φ a set of $\neg$-clauses over Var, $[\![\mathsf{diag}_\Phi]\!]$ is exactly the set of satisfying assignments of Φ.*

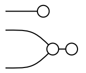
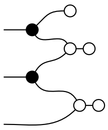
Proof. We continue the notation from the paragraph above. A tuple $(x_1, \ldots, x_m) \in \mathbb{B}^m$ is in $[\![\mathsf{diag}_\Phi]\!]$ precisely when $A\begin{pmatrix} x_1 \\ \vdots \\ x_m \end{pmatrix} \leq 0$, where A is the Boolean matrix representing diag_Φ. By the definition of A as $A_{ij} = 1$ if and only if $x_j \in \varphi_i$, this entails $(A_{i1} \wedge x_1) \wedge \cdots \wedge (A_{im} \wedge x_m)$

is equivalent to φ_i. So $A\begin{pmatrix}x_1\\\vdots\\x_m\end{pmatrix} \leq 0$ if and only if $\varphi_i(x_1,\ldots,x_m) \leq 0$, for $i = 1,\ldots,n$. In other words, $A\begin{pmatrix}x_1\\\vdots\\x_m\end{pmatrix} \leq 0$ if and only if $(x_1,\ldots,x_m)$ satisfyies Φ. □

Example 5.5.12. Let $\mathit{Var} = \{x_1, x_2, x_3\}$, and consider the following two sets of $\neg$-clauses:

$$\begin{aligned}\Gamma \quad &= \{\{x_1\}, \{x_2, x_3\}\}\\ \Phi \quad &= \{\{x_1\}, \{x_1, x_2\}, \{x_2, x_3\}\}\end{aligned}$$

They have the same set of satisfying assignments $A = \{000, 010, 001\}$; their associated pre-normal form diagrams are

$\mathsf{diag}_\Gamma =$  $\qquad \mathsf{diag}_\Phi =$ 

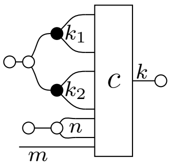
which verify $[\![\mathsf{diag}_\Gamma]\!] = [\![\mathsf{diag}_\Phi]\!] = A$. Moreover, Φ contains the 'redundant' $\neg$-clause $\{x_1, x_2\}$ because the clause $\{x_1\}$ already implies that $\{x_1, x_2\}$ is satisfied (if $\neg x_1 = 1$, then $\neg x_1 \vee \neg x_2 = 0$), while Γ contains no such redundancies. Indeed Γ is the smallest set of $\neg$-clauses whose set of satisfying assignments is A, and we shall see that such diag_Γ is in normal form.

Now we are ready to prove Lemma 5.5.8

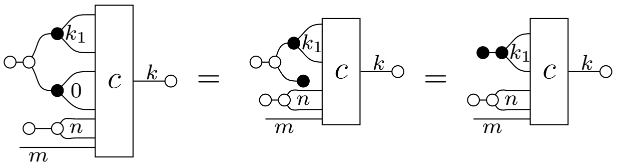
Proof. By now, Proposition 5.5.7 establishes that every diagram $d\colon m \to 0$ is equivalent to some existential canonical form diagram d', and such d' differs from pre-normal form only in that it may possibly be pre-composed with multiple cups, namely ○-◁⊏. So in order to show that every morphism of type $m \to 0$ is equal to some pre-normal form diagram, it suffices to prove that all existential canonical form diagrams can be transformed to pre-normal form diagrams by eliminating those pre-composed ○-◁⊏ one by one.

Formally, we prove by induction on the number of ○-◁⊏ in existential canonical form diagrams. We denote the number of ○-◁⊏ in diagram d by $\#_{○\text{-}◁⊏}(d)$. Let $d\colon m \to 0$ be for existential canonical form. For the base case, if $\#_{○\text{-}◁⊏}(d) = 0$, then d is already of pre-normal form.

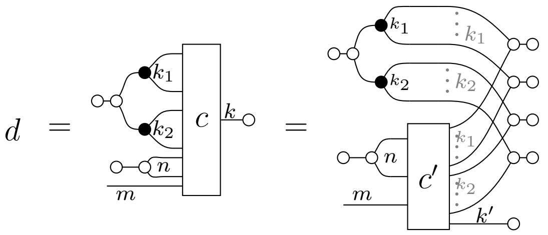
For the induction step, suppose $\#_{○\text{-}◁⊏}(d) = n + 1$, and we have the induction hypothesis (IH): for all existential canonical form morphisms e with $\#_{○\text{-}◁⊏}(e) \leq n$, there exists a pre-normal form diagram e' such that $e = e'$. Below we eliminate the first ○-◁⊏ from the top, such that for the resulting diagram one can simply apply IH. Without loss of generality we suppose d is of the following form, where k_1, k_2, k, n are natural numbers.



In particular, we assume that this presentation 'exhausts' all −●◁⊏ on the right-hand-side of the top ○◁⊏: for each wire of the $k_1 + k_2$ wires, there is no —●⊏ post-composed with them. This is always possible because c is a matrix diagram. Before moving on we solve one simple side case. If (at least) one of k_1 and k_2 is 0, then the top ○◁⊏ can be eliminated easily. For instance, if $k_2 = 0$, then



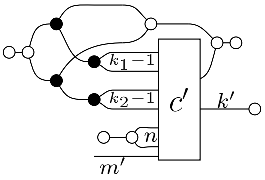
So below we assume that $k_1, k_2 \geq 1$. In this case, we claim that such d is always equivalent to a diagram on the right-hand-side of (5.30), where c' is some matrix diagram:



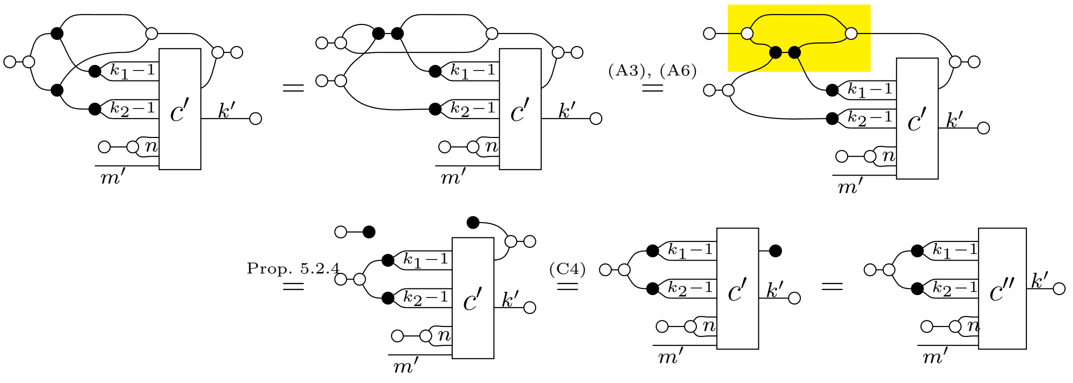
(5.30)

To prove this claim, the key is to study whether those k_1 wires and k_2 wires (on the right ports of the —●⊏) are connected with each other.

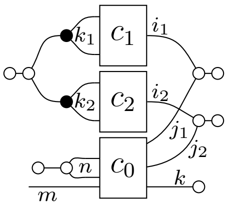
First, suppose that one of the k_1 wires and one of the k_2 wires are connected (as inputs to the matrix diagram c, see Definition 5.3.5). Without loss of generality we assume the top wires in each are connected. Then d is equivalent to



This means that there is a *loop* in the diagram, which is highlighted yellow as below. As a consequence, we can 'delete' these two connected wires without affecting the other $(k_1 - 1) + (k_2 - 1)$ wires, and the resulting diagram is also of pre-normal form.



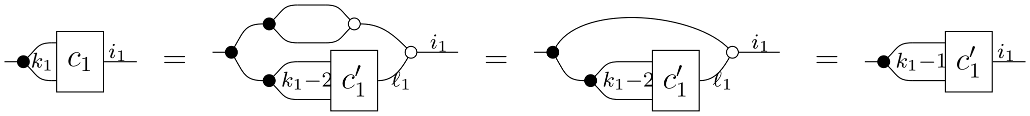
One can repeat this procedure until there are no two wires from the k_1 and k_2 wires respectively that are connected. Then the resulting diagram, by Proposition 5.3.15, can be decomposed as



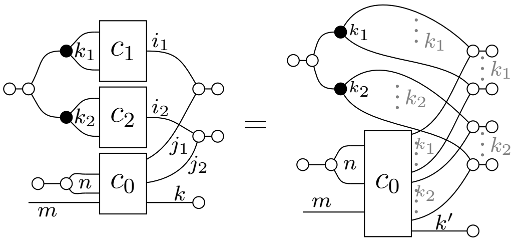
(5.31)

where c_0, c_1, c_2 are all matrix diagrams, and c_1, c_2 consist only of ⊐○— and ○— (namely no —●⊏, —●, nor ○—).

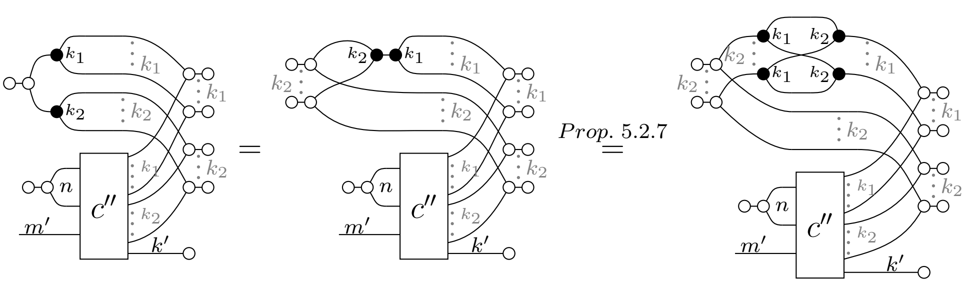
Next, we consider if two of the k_i wires are connected, for one of $i = 1, 2$. Without loss of generality, suppose two wires among the k_1 wires are connected, then indeed these two wires are 'redundant' in the sense that the diagram is equivalent to one which has only $k_1 - 1$ copies:

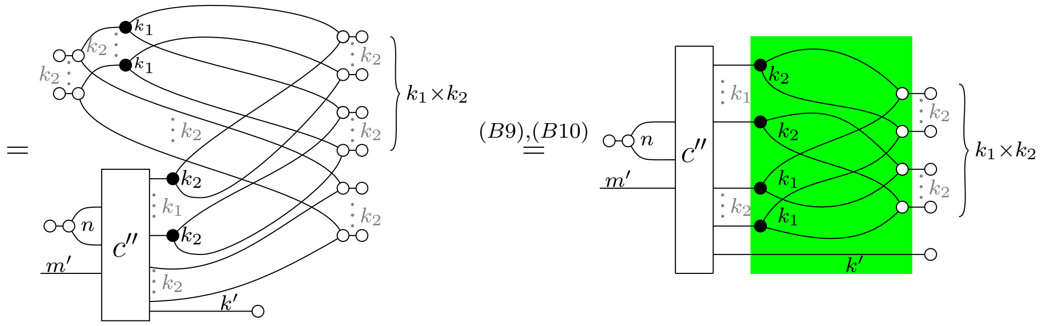


where the second step holds by Proposition 5.2.1, which proves in particular that (D2) can be strengthened to an equality. One can repeat this procedure until either $k_i < 2$, or any two wires in the k_i wires are not connected, for $i = 1, 2$. In both cases the c_1 and c_2 boxes can be 'opened':



So far we have proved (5.30), and we are now ready to eliminate the top ○○⊏, using Proposition 5.2.7 and Frobenius equations:





Note that the last diagram is of existential canonical form because the green part is a matrix diagram, whose composition with the matrix diagram c'' is again equivalent to a matrix diagram, by matrix completeness (Proposition 5.3.11). Moreover, it now has only n-many leftmost ○○⊏; thus, by the induction hypothesis it is equivalent to a pre-normal form diagram. □

5.5.3 Normal form and completeness

Recall that for completeness we need to select one representative diagram – in normal form – for each semantic object. Similar to the completeness proof of monotone functions over $\mathbb{B}$ (Subsection 5.4), we use sets of $\neg$-clauses as an intermediate step and choose such representative to be the least set of $\neg$-clauses among those with a given downward closed subset of $\mathbb{B}^m$ as their sets of satisfying assignments. The existence of such least set of $\neg$-clauses is guaranteed by Corollary 5.4.14. The proof is outlined as follows:

$$\begin{aligned}
& \text{pre-normal form diagram } d\colon m \to 0 \\
\Longrightarrow\ & [\![d]\!] \subseteq \mathbb{B}^m \text{ is downward closed} \\
\Longrightarrow\ & \text{there is a least set of } \neg\text{-clauses whose set of solutions is } [\![d]\!] \\
\Longrightarrow\ & \text{normal form diagram } d' \text{ that corresponds to the least set of } \neg\text{-clauses} \\
\Longrightarrow\ & d' = d
\end{aligned}$$

The normal form diagrams are defined as the diagrammatic counterpart of minimal sets of $\neg$-clauses.

Definition 5.5.13. Suppose $d\colon m \to 0$ is a pre-normal form diagram with representing matrix D. We say d is of *normal form* if there are no two distinct rows i, j of D such that $A_{ik} \leq A_{jk}$ for all $k = 1, \ldots, m$.

Note that swapping two rows in the representing matrix D of diagram d does not change clause_d (the *set* of $\neg$-clauses generated by d), so what we really care about is normal form diagrams up to permutation of the rows of their representing matrices. Let d and e be two normal form diagrams with representing $n \times m$ matrices D and E, respectively. We say d and e are *equivalent up to commutativity* if D is E with its rows permuted: there is some permutation $\tau\colon \{1, \ldots, n\} \to \{1, \ldots, n\}$ such that $D_{i-} = E_{\tau(i)-}$ for all $i \in \{1, \ldots, n\}$.

Example 5.5.14. Consider the following pre-normal form diagrams:

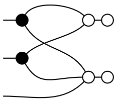
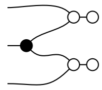
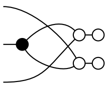
$c =$  $d =$  $e =$ 

Diagram c is not of normal form because it represents the Boolean matrix $C = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 1 & 1 \end{pmatrix}$ which satisfies $C_{1k} \leq C_{2k}$ for all $k = 1, 2, 3$. D and E are both of normal form, representing matrices $\begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix}$ and $\begin{pmatrix} 0 & 1 & 1 \\ 1 & 1 & 0 \end{pmatrix}$ respectively, thus d and e equivalent up to commutativity.

The next result state the connection between the three key concepts in the completeness proof.

Proposition 5.5.15. *There is a 1-1 correspondence between the following three sets:*

1. *downward closed subsets of* $\mathbb{B}^m$*;*
2. *minimal sets of* $\neg$*-clauses over* $Var = \{x_1, \ldots, x_m\}$*;*
3. *normal form diagrams of type* $m \to 0$*, modulo equivalence up to commutativity.*

Proof. **(1) ≅ (2):** It suffices to define two functions between (1) and (2) which are injective and inverses to each other. Note that a subset of $\mathbb{B}^m$ is downward closed if and only if it is the set of satisfying assignments to some set of $\neg$-clauses (Lemma 5.5.16). Starting from a downward closed subset $A \subseteq \mathbb{B}^m$, one have a least set of $\neg$-clauses whose set of satisfying assignments is A (Corollary 5.4.14). Inversely, every set of $\neg$-clauses uniquely determines a downward closed subset of $\mathbb{B}^m$, namely the set of its satisfying assignments. It follows immediately that these two functions are inverses to each other.
(2) ≅ (3): We notice that when restricted to minimal sets of $\neg$-clauses (resp. normal form diagrams), the images of $\mathsf{diag}_{(\cdot)}$ (resp. $\mathsf{clause}_{(\cdot)}$) are normal form diagrams (resp. minimal sets of $\neg$-clause). Moreover, two normal form diagrams have the same $\mathsf{diag}_{(\cdot)}$-image precisely when they are equivalent up to commutativity. Thus $\mathsf{diag}_{(\cdot)}$ and $\mathsf{clause}_{(\cdot)}$ witness the isomorphism. □

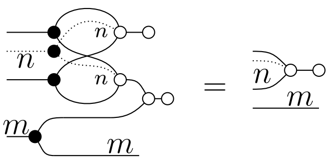
Lemma 5.5.16. *A subset of* $\mathbb{B}^{Var}$ *is downward closed if and only if it is the satisfying assignment of some set of* $\neg$*-clauses over* Var*.*

Proof. The 'if' direction is straightforward. A clause $\{x_{p_1}, \ldots, x_{p_{k_i}}\}$ represents the formula $x_{p_1} \wedge \cdots \wedge x_{p_{k_i}} = 0$ or, equivalently, $\neg x_{p_1} \vee \cdots \vee \neg x_{p_{k_i}} = 1$. It is clear that the set of satisfying assignments of such a clause is downward closed.

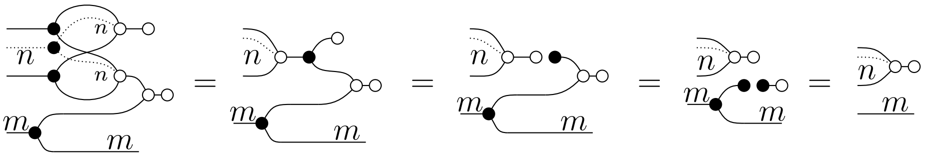
For the 'only if' direction, let S be a downward closed subset of $\mathbb{B}^{Var}$. Let Γ be the set of all clauses whose set of satisfying assignments contain S (note that this may be the set containing the empty clause if S is empty). Then clearly, all $\mathbf{s} \in S$ satisfy Γ. Suppose that there exists some $\mathbf{t} \notin S$ which also satisfy Γ. Then write $\varphi = \{i_1, \ldots, i_p\} \subseteq \{1, \ldots, m\}$ for the set of components i_k at which there exists $\mathbf{s}_k \in S$ such that $\mathbf{s}_k(i_k) = 0 = \neg\mathbf{t}(i_k)$ (note that there is at least one such component, otherwise $\mathbf{t}$ would be in S). By construction, elements of S satisfy φ while $\mathbf{t}$ does not. This implies that φ must be in Γ, so $\mathbf{t}$ should satisfy φ too, a contradiction. □

Having established the connection between semantic objects and normal form diagrams, we show that indeed every Syn-morphism is equal to one in normal form. Crucially we need the following lemma to turn a pre-normal form into one of normal form. This diagrammatic operation corresponds to removing redundant clauses in a set of ¬-clauses: if some ¬-clause σ is strictly larger than another γ, then one can drop σ from Σ and the resulting set of ¬-clauses has the same set of satisfying assignments with the original Σ.

Lemma 5.5.17. *For arbitrary natural numbers m, n,*

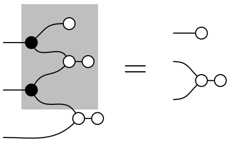


Proof.



Here the first step uses Proposition 5.2.8; the second and third steps use Proposition 5.2.2. □

Example 5.5.18. Recall the two diagrams diag_Γ and diag_Φ from Example 5.5.12, and we note that diag_Φ is in normal form while diag_Γ is not. We can apply Lemma 5.5.17 (to the grey block, instantiated with $m = n = 1$) to rewrite diag_Γ (on the lhs) into the normal form diagram diag_Φ (on the rhs):



Lemma 5.5.19 (Normal form)**.** *Every diagram $d\colon m \to 0$ is equal to a diagram in normal form.*

Proof. Given a pre-normal form diagram $d\colon m \to 0$ which is *not* in normal form. By Definition 5.5.13, this means that in the $n \times m$ Boolean matrix D represented by d, there exist two rows i and j such that $D_{ik} \leq D_{jk}$ for all $k = 1, \ldots, m$. Diagrammatically, this means that there is the same pattern as on the lhs of Lemma 5.5.17: there exists some —○ (*i.e.* the i-th one) such that all the input wires that are connected with this —○ is also connected with some other —○ (*i.e.* the j-th one). In this case, we apply Lemma 5.5.17 (from left to right) and turn diagram d into a pre-normal form diagram e, whose representing matrix E is of type $(n-1) \times m$, and it is exactly D but without the i-th row.

Since D is finite, such pattern disobeying the definition of normal form can only happen finitely many times, each can be solved by the above procedure. The resulting diagram is of normal form and equivalent to the original diagram d. $\square$

Finally we are ready to prove the completeness part of Theorem 5.5.21. We start with the following completeness statement (involving only equalities), which follows from Proposition 5.5.15 and Lemma 5.5.19.

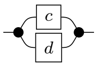
Lemma 5.5.20. *For any diagrams $c, d\colon m \to n$, if $[\![c]\!] = [\![d]\!]$, then $c = d$.*

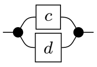
Proof. Given Lemma 5.5.4 it suffices to prove this statement for morphisms with codomains 0. Let $c, d\colon m \to 0$ be two diagrams satisfying $[\![c]\!] = [\![d]\!] = A$, for some downward closed subset A of $\mathbb{B}^{Var}$ where $Var = \{x_1, \ldots, x_m\}$. By Proposition 5.5.15, A is the set of satisfying assignments of some minimal set of clauses, say Γ. By Lemma 5.5.19, c and d are equivalent to some normal form diagrams, say $\hat{c}$ and $\hat{d}$, respectively. Moreover, $\hat{c}$ and $\hat{d}$ verify $\mathsf{clause}_{\hat{c}} = \mathsf{clause}_{\hat{d}} = \sigma$.

Note that every set of $\neg$-clauses corresponds to a diagram in pre-normal form; when this set of $\neg$-clauses is minimal, the corresponding diagram is in normal form, by Proposition 5.5.15. Now $\hat{c}$ and $\hat{d}$ are both normal form diagrams interpreted as A, so they are both the normal form diagram uniquely determined by Γ, up to *SATA* equality. This means that $\hat{c} = \hat{d}$, thus $c = \hat{c} = \hat{d} = d$. $\square$

Now we are ready to prove completeness.

Theorem 5.5.21. *For arbitrary $\mathbb{SATA}$-morphisms c and d, $c \leq_{\mathbb{SATA}} d$ if and only if $[\![c]\!] \subseteq [\![d]\!]$.*

Proof. We assume that $[\![c]\!] \subseteq [\![d]\!]$. By Lemma 5.5.3*(ii)*, $[\![$ 

$]\!] = [\![c]\!]$. By Lemma 5.5.20, this implies 

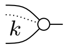
$= c$, thus $c \leq d$ according to Lemma 5.5.3 *(i)*. $\square$

5.6 Application: propositional definite logic programs

So far we have established the theory *SATA* that presents monotone relations over finite Boolean algebras. In this subsection we show its application in propositional definite logic programs. Note the two restrictions here to the *propositional* and *definite* case, which corresponds to the *finiteness* and *monotonicity* in our theory.

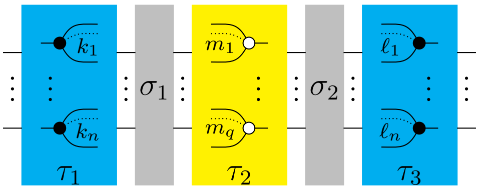
We show how to represent two important operators — immediate consequence operator and consequence operator — for logic programs diagrammatically using $\mathbb{SATA}$. In particular, the axiomatisation *SATA* enables one to reason about the (in)equivalence between such operators of two logic programs.

Let us fix a set At of atoms and a propositional definite logic program $\mathbb{L}$ over At.

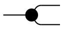
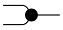
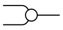
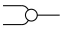
We first consider immediate consequence operator. The key observation is the intuitive reading of clause $b_1, \ldots, b_k \to a$ as $b_1 \wedge \cdots \wedge b_k \leq a$, thus representable as the diagram 

.

Definition 5.6.1. The *immediate consequence diagram* $\mathsf{T}_{\mathbb{L}}$ for program $\mathbb{L}$ is an $n \to n$ morphism defined as follows, where the i-th wire on the left and right ports stands for atom a_i, respectively, for $i = 1, \dots, n$.



(5.32)

The two blue blocks τ_1 and τ_3 consist respectively of copiers  and cocopiers : k_i and ℓ_i are the numbers of appearance of atom a_i in all the bodies and heads of $\mathbb{L}$-clauses, respectively. The yellow block τ_2 is the parallel composition of , where q is the number of clauses in $\mathbb{L}$, and for each clause ψ_j, m_j is the size of $\mathsf{body}(\psi_j)$. The grey blocks σ_1 and σ_2 consist of appropriately many that change the wire orders to match the domains and codomains of the blue and yellow blocks. In particular, the i-th wire on the left port (*resp.* on the right port) is connected to the j-th  if $a_i \in \mathsf{body}(\psi_j)$ (*resp.* $a_i = \mathsf{head}(\psi)$).

Note that the interpretation $[\![\mathsf{T}_{\mathbb{L}}]\!]$ is not exactly the operator $\mathbf{T}_{\mathbb{L}}$: the former is a monotone relation while the latter is a monotone function. Nevertheless we recall that a monotone relation $R \subseteq X \times Y$ *represents* a monotone function $f \colon X \to Y$ if for arbitrary $x \in X$ and $y \in Y$, $(x, y) \in R$ if and only if $f(x) \leq y$. Crucially, if monotone relations R and S represent monotone functions f and g respectively, then $f = g$ if and only if $R = S$.

Proposition 5.6.2. *The monotone relation* $[\![\mathsf{T}_{\mathbb{L}}]\!]$ *represents the operator* $\mathbf{T}_{\mathbb{L}}$.

Proof. We show that for arbitrary interpretations $\mathcal{I}$, $\mathcal{J}$ on At, $(\mathcal{I}, \mathcal{J}) \in [\![\mathsf{T}_{\mathbb{L}}]\!]$ if and only if $\mathbf{T}_{\mathbb{L}}(\mathcal{I}) \leq \mathcal{J}$. We start from the left-hand-side and use the notation from Definition 5.6.1,

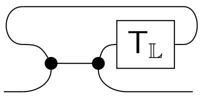
$$\begin{aligned}
& (\mathcal{I}, \mathcal{J}) \in [\![\mathsf{T}_{\mathbb{L}}]\!] \\
\iff & \exists \mathcal{K}_1 \in \mathbb{B}^{\ell_1}, \dots, \mathcal{K}_n \in \mathbb{B}^{\ell_n} \colon (\mathcal{I}, (\mathcal{K}_1, \dots, \mathcal{K}_n)) \in [\![\tau_1 \mathbin{;} \sigma_1 \mathbin{;} \tau_2 \mathbin{;} \sigma_2]\!], ((\mathcal{K}_1, \dots, \mathcal{K}_n), \mathcal{J}) \in [\![\tau_3]\!] \\
\iff & \exists \mathcal{K}_1, \dots, \mathcal{K}_n \colon (\mathcal{I}, (\mathcal{K}_1, \dots, \mathcal{K}_n)) \in [\![\tau_1 \mathbin{;} \sigma_1 \mathbin{;} \tau_2 \mathbin{;} \sigma_2]\!], \mathcal{K}_i(j_i) \leq \mathcal{J}(i) \text{ for } \forall i = 1, \dots, n, j_i = 1, \dots, \ell_i \\
\iff & \text{for each } \mathbb{L}\text{-clause } \psi \equiv a_{i_1}, \dots, a_{i_r} \to a_{i_s}, \mathcal{I}(i_1) \wedge \cdots \wedge \mathcal{I}(i_r) \leq \mathcal{J}(i_s) \\
\iff & \text{for each } \mathbb{L}\text{-clause } \psi \equiv a_{i_1}, \dots, a_{i_r} \to a_{i_s}, \text{if } \mathcal{I}(i_1) = \cdots = \mathcal{I}(i_r) = 1, \text{then } \mathcal{J}(i_s) = 1 \\
\iff & \mathbf{T}_{\mathbb{L}}(\mathcal{I}) \leq \mathcal{J}
\end{aligned}$$

Therefore the monotone relation $[\![\mathsf{T}_{\mathbb{L}}]\!]$ represents the monotone function $\mathbf{T}_{\mathbb{L}}$. □

Based on this, we can compositionally represent the consequence operator $\mathbf{T}_{\mathbb{L}}$ as $\mathsf{T}_{\mathbb{L}}$ plus some feedback loops that play the role of least fixed point.

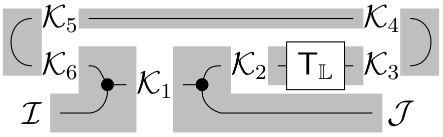
Definition 5.6.3. The *consequence diagram* for program $\mathbb{L}$ is a Syn-morphism $\mathsf{C}_{\mathbb{L}} \colon n \to n$

defined on the right, where $\mathsf{T}_{\mathbb{L}}$ is the immediate consequence diagram from Definition 5.6.1.



Proposition 5.6.4. *The monotone relation $\llbracket \mathsf{C}_{\mathbb{L}} \rrbracket$ represents the operator $\mathbf{C}_{\mathbb{L}}$.*

Proof. We show that for arbitrary interpretations $\mathcal{I}, \mathcal{J}$ on At, $(\mathcal{I}, \mathcal{J}) \in \llbracket \mathsf{C}_{\mathbb{L}} \rrbracket$ if and only if $\mathbf{C}_{\mathbb{L}}(\mathcal{I}) \leq \mathcal{J}$.



(5.33)

By the definition of the interpretation $\llbracket \cdot \rrbracket$, $(\mathcal{I}, \mathcal{J}) \in \llbracket \mathsf{C}_{\mathbb{L}} \rrbracket$ if and only if there exist $\mathcal{K}_i$ for $i = 1, \ldots, 6$ such that: $\mathcal{I}, \mathcal{K}_6 \leq \mathcal{K}_1$; $\mathcal{K}_1 \leq \mathcal{K}_2, \mathcal{J}$; $(\mathcal{K}_2, \mathcal{K}_3) \in \llbracket \mathsf{T}_{\mathbb{L}} \rrbracket$; $\mathcal{K}_3 \wedge \mathcal{K}_4 \leq \mathbf{0}$; $\mathcal{K}_5 \leq \mathcal{K}_4$; $\mathbf{1} \leq \mathcal{K}_5 \vee \mathcal{K}_6$. See (5.33) for an illustration of these $\mathcal{K}_i$. We can simplify this condition by a sequence of equivalent conditions on $\mathcal{I}, \mathcal{J}$ and those $\mathcal{K}_i$:

$$
\begin{aligned}
& \mathcal{I}, \mathcal{K}_6 \leq \mathcal{K}_2, \mathcal{J}; (\mathcal{K}_2, \mathcal{K}_3) \in \llbracket \mathsf{T}_{\mathbb{L}} \rrbracket ; \mathcal{K}_3 \wedge \mathcal{K}_4 \leq \mathbf{0}; \mathcal{K}_5 \leq \mathcal{K}_4; \mathbf{1} \leq \mathcal{K}_5 \vee \mathcal{K}_6 \\
\Longleftrightarrow\ & \mathcal{I}, \mathcal{K}_6 \leq \mathcal{K}_2, \mathcal{J}; (\mathcal{K}_2, \mathcal{K}_3) \in \llbracket \mathsf{T}_{\mathbb{L}} \rrbracket ; \mathcal{K}_3 \leq \mathcal{K}_6 \\
\Longleftrightarrow\ & \mathcal{I}, \mathcal{K}_3 \leq \mathcal{K}_2, \mathcal{J}; (\mathcal{K}_2, \mathcal{K}_3) \in \llbracket \mathsf{T}_{\mathbb{L}} \rrbracket \\
\Longleftrightarrow\ & \mathcal{I}, \mathcal{K}_3 \leq \mathcal{K}_2, \mathcal{J}; \mathbf{T}_{\mathbb{L}}(\mathcal{K}_2) \leq \mathcal{K}_3 \\
\Longleftrightarrow\ & \mathcal{I}, \mathbf{T}_{\mathbb{L}}(\mathcal{K}_2) \leq \mathcal{K}_2, \mathcal{J}
\end{aligned}
$$

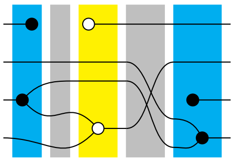
That is to say, $(\mathcal{I}, \mathcal{J}) \in \llbracket \mathsf{C}_{\mathbb{L}} \rrbracket$ if and only if there exists $\mathcal{K}$ such that $\mathcal{I}, \mathbf{T}_{\mathbb{L}}(\mathcal{K}) \leq \mathcal{K}, \mathcal{J}$. We claim that this is equivalent to that $\mathcal{J} \geq \mu U.\mathcal{I} \vee \mathbf{T}_{\mathbb{L}}(U)$, where $\mu U.F(U)$ denotes the least fixed point of a monotone function F on some complete lattice. On one hand, if $\mathcal{J} \geq \mu U.\mathcal{I} \vee \mathbf{T}_{\mathbb{L}}(U)$, then let $\mathcal{K}$ be $\mu U.\mathcal{I} \vee \mathbf{T}_{\mathbb{L}}(U)$. By the definition of fixed points, $\mathcal{I} \vee \mathbf{T}_{\mathbb{L}}(\mathcal{K}) = \mathcal{K}$, we know that $\mathcal{I} \leq \mathcal{K}$ and $\mathbf{T}_{\mathbb{L}}(\mathcal{K}) \leq \mathcal{K}$. So $\mathcal{I}, \mathbf{T}_{\mathbb{L}}(\mathcal{K}) \leq \mathcal{J}$ as well. On the other hand, suppose there exists $\mathcal{K}$ such that $\mathcal{I}, \mathbf{T}_{\mathbb{L}}(\mathcal{K}) \leq \mathcal{K}, \mathcal{J}$, then this is equivalent to $\mathcal{I} \vee \mathbf{T}_{\mathbb{L}}(\mathcal{K}) \leq \mathcal{K}, \mathcal{J}$. $\mathcal{I} \vee \mathbf{T}_{\mathbb{L}}(\mathcal{K}) \leq \mathcal{K}$ means that $\mathcal{K}$ is a pre-fixed point for $\mathcal{I} \vee \mathbf{T}_{\mathbb{L}}(-)\colon \mathbb{B}^{At} \to \mathbb{B}^{At}$, which, by Knaster-Tarski theorem, entails $\mu U.\mathcal{I} \vee \mathbf{T}_{\mathbb{L}}(U) \leq \mathcal{K}$. Therefore, using the fact that $\mathbf{T}_{\mathbb{L}}$ is monotone, we have

$$\mathcal{J} \geq \mathcal{I} \vee \mathbf{T}_{\mathbb{L}}(\mathcal{K}) \geq \mathcal{I} \vee \mathbf{T}_{\mathbb{L}}(\mu U.\mathcal{I} \vee \mathbf{T}_{\mathbb{L}}(U)) = \mu U.\mathcal{I} \vee \mathbf{T}_{\mathbb{L}}(U) \tag{5.34}$$

Now note that $\mu U.\mathcal{I} \vee \mathbf{T}_{\mathbb{L}}(U)$ is exactly $\mathbf{C}_{\mathbb{L}}(\mathcal{I})$ (Lemma **??**), so we have $(\mathcal{I}, \mathcal{J}) \in \llbracket \mathsf{C}_{\mathbb{L}} \rrbracket$ if and only if $\mathbf{C}_{\mathbb{L}}(\mathcal{I}) \leq \mathcal{J}$, which means that $\llbracket \mathsf{C}_{\mathbb{L}} \rrbracket$ represents the consequence operator $\mathbf{C}_{\mathbb{L}}$. □

Example 5.6.5. Consider the program $\mathbb{P} = \{\to a; b \to d; c \to d; c, d \to b\}$ over $At = \{a, b, c, d\}$. Its least Herbrand model $\mathcal{M}^H_{\mathbb{P}}$ is $\{a\}$. The immediate consequence operator $\mathbf{T}_{\mathbb{P}}$ maps, for instance, $\{c\}$ to $\{a, d\}$. The consequence operator $\mathbf{C}_{\mathbb{P}}$ maps, for instance, $\{c\}$ to $\{a, b, c, d\}$.

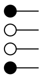
Now let us turn to its diagrammatic counterpart. Its immediate consequence diagram $\mathsf{T}_{\mathbb{P}}$ is given below left, coloured as in Definition 5.6.1.

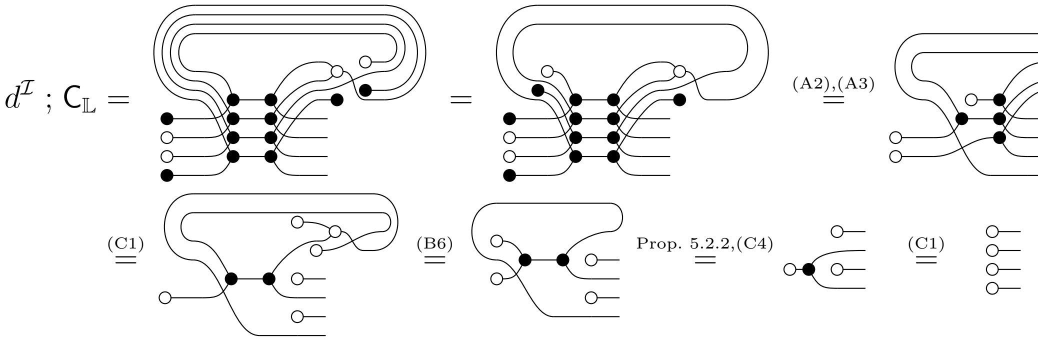


Now we are ready to present the two main results for logic programs. First, *SATA* provides a diagrammatic calculus to compute $\mathbf{C}_{\mathbb{L}}$, in particular to calculate the least Herbrand model as $\mathbf{C}_{\varnothing}$. One can represent an interpretation $\mathcal{I} \in \mathbb{B}^{At}$ as a diagram $d^{\mathcal{I}} := d_1^{\mathcal{I}} \otimes \cdots \otimes d_n^{\mathcal{I}}$ of type $0 \to n$, where each $d_i^{\mathcal{I}} =$ ●— if $\mathcal{I}(i) = 0$, and $d_i^{\mathcal{I}} =$ ○— if $\mathcal{I}(i) = 1$.

Theorem 5.6.6. $\mathsf{C}_{\mathbb{L}}(\mathcal{I}) = \mathcal{J}$ *if and only if* $d^{\mathcal{I}} \mathbin{;} \mathsf{C}_{\mathbb{L}} = d^{\mathcal{J}}$*.*

Proof. By Theorem 5.5.21, $d^{\mathcal{I}} \mathbin{;} \mathsf{C}_{\mathbb{L}} = d^{\mathcal{J}}$ if and only if $[\![d^{\mathcal{I}} \mathbin{;} \mathbf{C}_{\mathbb{L}}]\!] = [\![d^{\mathcal{J}}]\!]$. Since $[\![d^{\mathcal{I}}]\!]$ and $[\![\mathsf{C}_{\mathbb{L}}]\!]$ represent $\mathcal{I}$ and $\mathbf{C}_{\mathbb{L}}$ respectively, $[\![d^{\mathcal{I}} \mathbin{;} \mathsf{C}_{\mathbb{L}}]\!] = [\![d^{\mathcal{I}}]\!] \mathbin{⨾} [\![\mathsf{C}_{\mathbb{L}}]\!]$ represents $\mathbf{C}_{\mathbb{L}}(\mathcal{I})$, by the compositionality of representations. Then $[\![d^{\mathcal{I}} \mathbin{;} \mathsf{C}_{\mathbb{L}}]\!] = [\![d^{\mathcal{J}}]\!]$ is equivalent to $\mathbf{C}_{\mathbb{L}}(\mathcal{I}) = \mathcal{J}$. □

Example 5.6.7. Continue Example 5.6.5. Let $\mathcal{I} = \{b, c\}$, then $\mathbf{C}_{\mathbb{L}}(\mathcal{I}) = \{a, b, c, d\}$, which we denote as $\mathcal{J}$. $\mathcal{I}$ is represented by the diagram  , and



The resulting diagram represents $\mathcal{J}$.

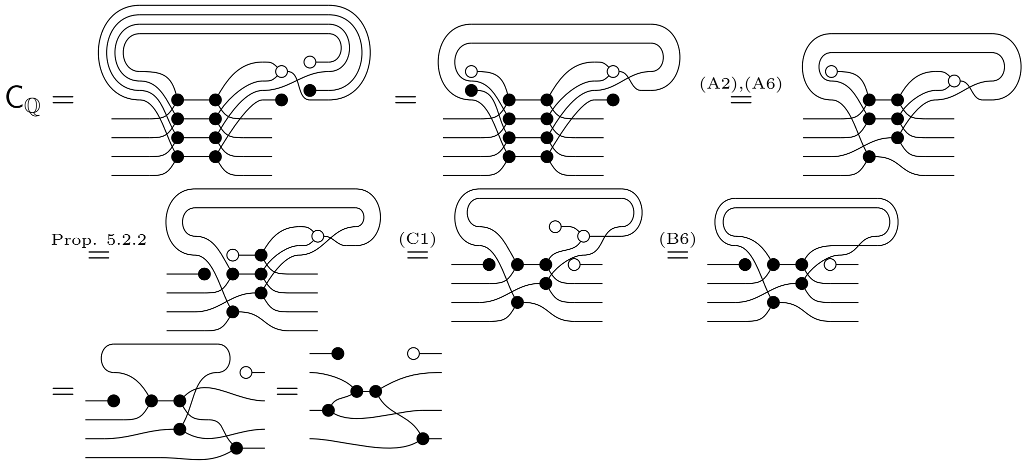
Second, we can turn the problem of the (in)equivalence of two logic programs into proving whether their consequence operator diagrams are equivalent. By Proposition 5.6.4 and Theorem 5.5.21, we have:

Theorem 5.6.8. *Given two logic programs* $\mathbb{L}_1$ *and* $\mathbb{L}_2$ *on* At*,* $\mathbf{C}_{\mathbb{L}_1} \geq \mathbf{C}_{\mathbb{L}_2}$ *as monotone functions* $\mathbb{B}^{At} \to \mathbb{B}^{At}$ *if and only if* $\mathsf{C}_{\mathbb{L}_1} \leq_{SATA} \mathsf{C}_{\mathbb{L}_2}$*.*

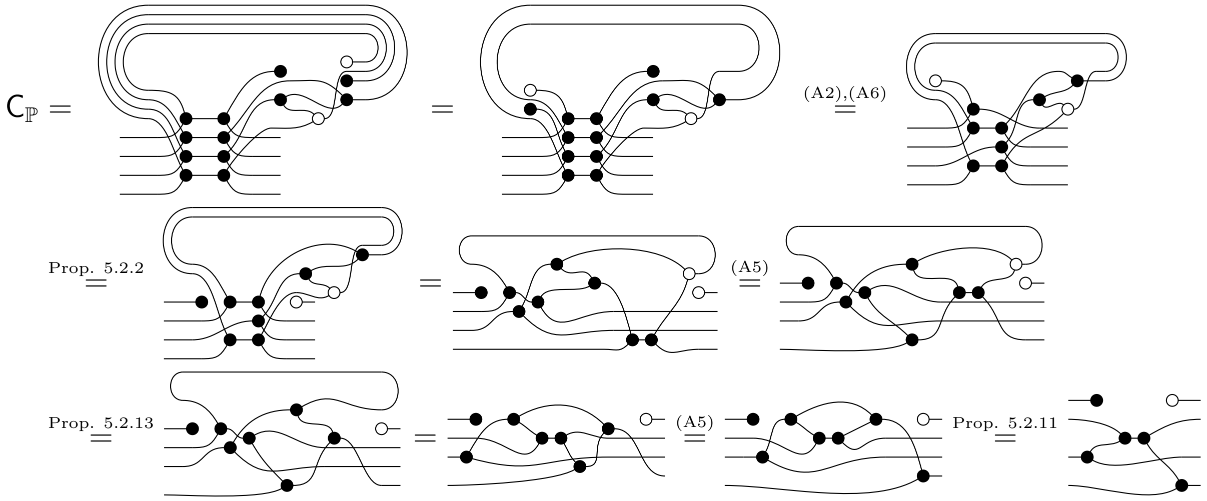
Proof. Suppose $\mathbf{C}_{\mathbb{L}_1} \geq \mathbf{C}_{\mathbb{L}_2}$, namely $\mathbf{C}_{\mathbb{L}_1}(a) \geq \mathbf{C}_{\mathbb{L}_2}(a)$ for all $a \in At$. This is is equivalent to saying that $\mathcal{R}(\mathbf{C}_{\mathbb{L}_1}) \subseteq \mathcal{R}(\mathbf{C}_{\mathbb{L}_2})$. Proposition 5.6.4 says that $[\![\mathsf{C}_{\mathbb{L}}]\!] = \mathcal{R}\mathbf{C}_{\mathbb{L}}$, thus $[\![\mathsf{C}_{\mathbb{L}_1}]\!] = \mathcal{R}(\mathbf{C}_{\mathbb{L}_1}) \subseteq \mathcal{R}(\mathbf{C}_{\mathbb{L}_2}) = [\![\mathsf{C}_{\mathbb{L}_2}]\!]$. Then completeness (Theorem 5.5.21) entails $\mathsf{C}_{\mathbb{L}_1} \leq \mathsf{C}_{\mathbb{L}_2}$. The reverse of the above reasoning also holds, where we only need soundness rather than completeness. □

It is worth observing that the completeness proof (see Section 5.5.3) is essentially an algorithm that rewrites an arbitrary Syn-morphism into a normal form diagram (Definition 5.5.13). This means that the procedure described in Theorem 5.6.8 to establish (in)equivalence of logic programs in terms of consequence operators is decidable. We illustrate such procedure with an example.

Example 5.6.9. Recall $\mathbb{P}$ from Example 5.6.5, and consider another program $\mathbb{Q} = \{\to a; a, b \to d; c \to b.\}$. We claim that $\mathbb{P}$ and $\mathbb{Q}$ are equivalent with respect to consequence operator, namely $\mathbf{C}_{\mathbb{P}} = \mathbf{C}_{\mathbb{Q}}$, and we show this by proving the equivalence $\mathsf{C}_{\mathbb{P}} = \mathsf{C}_{\mathbb{Q}}$ of their consequence diagrams. On one hand,

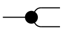
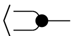
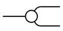
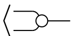


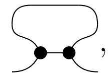
On the other hand,



Thus $\mathsf{C}_{\mathbb{P}} = \mathsf{C}_{\mathbb{Q}}$, which entails $\mathbf{C}_{\mathbb{P}} = \mathbf{C}_{\mathbb{Q}}$ by Theorem 5.5.21.

As a remark we mention another possibility of defining the semantics to interpret Syn in the category Rel of relations, say via a functor $\langle\cdot\rangle\colon \mathsf{Syn} \to \mathsf{Rel}$, where:

$\langle$$\rangle = \{(x,(x,x)) \mid x \in \mathbb{B}\}$ $\langle$$\rangle = \{((x_1,x_2),y) \mid x_1 \vee x_2 = y \in \mathbb{B}\}$
$\langle$$\rangle = \{(x,(y_1,y_2)) \mid x = y_1 \wedge y_2\}$ $\langle$$\rangle = \{((x_1,x_2),y) \mid x_1 \wedge x_2 = y\}$

Under this semantics, the immediate consequence diagram $\mathsf{T}_{\mathbb{L}}$ has the correct semantics as $\{(\mathcal{I},\mathcal{J}) \mid \mathbf{T}_{\mathbb{L}}(\mathcal{I}) = \mathcal{J}\}$. However, $\mathsf{C}_{\mathbb{L}}$ — an intuitive candidate to represent consequence operators — is not interpreted as $\{(\mathcal{I},\mathcal{J}) \mid \mathbf{C}_{\mathbb{L}}(\mathcal{I}) = \mathcal{J}\}$. In particular, the trivial program $\mathbb{L} = \{a \to a\}$ has $\mathsf{C}_{\mathbb{L}} =$ , and $\langle \mathsf{C}_{\mathbb{L}} \rangle = \{(x,y) \mid x,y \in \mathbb{B}, x \leq y\} \neq \{(x,x) \mid x \in \mathbb{B}\} = \mathbf{C}_{\mathbb{L}}$. Looking for an appropriate diagrammatic language and interpretation in Rel is important to the purpose of generalising the current result to logic programs with negation and their semantics, because their immediate consequence operators are *not* monotone in general. Such an interpretation together with the least fixed point constructions in some canonical semantics for general logic programs seems to imply that diagram similar to $\mathsf{C}_{\mathbb{P}}$ would represent the semantics. We leave exploration of this idea to future work.

Chapter 6

Further directions

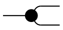
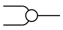
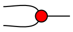
In this thesis we have explored coalgebraic and algebraic semantics for logic programming and its quantitative variants. Crucially, the study aims at a uniform framework such that the semantics of those variants can be captured in a modular way. Admittedly, this thesis is a first step towards a full story of these logic programming paradigms in terms of category theory, and there are a few future directions.

Bialgebraic semantics. First there is an immediate story following the two topics of this thesis. Given the coalgebraic and algebraic treatment of logic programming, one natural question is about their interaction. To the best of our knowledge, such interaction for LP is encoded as a bialgebraic semantics for definite logic programs [BZ15], where the algebraic and coalgebraic characterisation of a system interact via a distributive law [Kli11]. To explore a similar approach for PLP and WLP, the first technical challenge would be to find the appropriate functors to encode the algebraic and coalgebraic types of the programs, such that there exists a suitable distributive law. This problem is non-trivial for distribution monads and multiset functors [Var03, VW06, Jac21]. We are not devoted to this problem in the thesis and leave it for future work, since our algebraic semantics is defined in terms of functors, and it is not immediate to see how it interacts with the coalgebraic semantics.

First-order case. One immediate next step is to extend the algebraic approach using functorial semantics to logic programs with variables. Admittedly, as far as we know, most application of PLP and WLP using variables are *function-free*: there is no function symbol in the signature, thus the Herbrand base is finite given a finite signature. Existing work on diagrammatic Lawvere theory could be a starting point for dealing with variables in the logic [BSZ18].

Diagrammatic PLP and WLP. Concerning the diagrammatic calculus, one immediate next step is to generalise the current calculus for definite logic programs to probabilistic and weighted case. For the former, we are already exploring a diagrammatic calculus based on $SATA$, whose arrows $m \to n$ are finite distributions of arrows $m \to n$ in $SATA$. We believe such categorical construction is a better approximation of the distribution semantics for PLP.

For the latter, one may start with looking at some specific semiring structure, in particular the min-plus semiring for shortest path problems. We suggest a diagrammatic calculus with

, , and  (together with their transposes), standing for copy, sum, and minimise. If we obtain such a complete calculus, one gain is that there is no need for extra specification of shortest path problems apart from the pictures: one can encode a given weighted directed graph using the diagrammatic language, and calculate the weight of the shortest path diagrammatically.

Answer set programming. Answer set programming (ASP) is logic programming based on so-called *stable model* semantics, dealing with clauses with negations [Lif19]. As an immediate consequence, the semantics of a program may consists of multiple stable models, thus the name 'answer set'. One interesting question is whether such stable model semantics could be captured functorially by choosing a suitable semantic category. Turning to its probabilistic variant probabilistic answer set programming (PASP) [CM20], since ASP programs have multiple models in general, the probability assigned to atoms in a PASP program is *imprecise*. To the best of our knowledge there is no published categorical treatment of such imprecise probabilities [ACDCT14]. In particular, it is worth exploring whether notions such as conditional probabilities and marginalisation in imprecise probability theory fits into the synthetic statistics framework [Fri20].

Inductive logic programming. Inductive logic programming (ILP) is a learning paradigm using logic programs as the underlying tools. To put it simple, given a logic program $\mathbb{L}$ as background knowledge together with sets of atoms representing positive and negative samples, the task is to learn a program extending $\mathbb{L}$ such that it admits all positive samples and rejects all negative ones [MDR94]. One interesting question is whether such learning can be phrased at the diagrammatic level. The topic of learning has attracted much attention in the applied category theory society, including string diagrammatic framework for neural networks [FST19, FS19, XM21], gradient descent [CGG^{+}22], supervised learning [FJ19], etc. In particular, we wonder whether a learner in ILP can be captured as certain lense [FJ19]. A derivative research question is to phrase automata learning [Ang87, vHSS17] in string diagram terms, based on the complete diagrammatic calculus for nodeterministic finite automata [PZ21]. Such a uniform high-level treatment of learning in logic programming and automata shall shed light on the similarity between these two fields of research.

Bibliography

[AB94] Krzysztof R Apt and Roland N Bol. Logic programming and negation: A survey. *The Journal of Logic Programming*, 19:9–71, 1994.

[ABW88] 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.

[AC09] Samson Abramsky and Bob Coecke. Categorical quantum mechanics. *Handbook of quantum logic and quantum structures*, 2:261–325, 2009.

[ACDCT14] Thomas Augustin, Frank PA Coolen, Gert De Cooman, and Matthias CM Troffaes. *Introduction to imprecise probabilities.* John Wiley & Sons, 2014.

[ALM09] Gianluca Amato, James Lipton, and Robert McGrail. On the algebraic structure of declarative programming languages. *Theoretical Computer Science*, 410(46):4626–4671, 2009.

[AM75a] Michael A Arbib and Ernest G Manes. Adjoint machines, state-behavior machines, and duality. *Journal of Pure and Applied Algebra*, 6(3):313–344, 1975.

[AM75b] Michael A Arbib and Ernest G Manes. A categorist's view of automata and systems. In *Category theory applied to computation and control*, pages 51–64. Springer, 1975.

[Ang87] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and computation*, 75(2):87–106, 1987.

[Awo10] Steve Awodey. *Category theory.* Oxford university press, 2010.

[BHP$^+$19] Filippo Bonchi, Joshua Holland, Robin Piedeleu, Paweł Sobociński, and Fabio Zanasi. Diagrammatic algebra: from linear to concurrent systems. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–28, 2019.

[BMR01] Roberto Bruni, Ugo Montanari, and Francesca Rossi. An interactive semantics of logic programming. *Theory and Practice of Logic Programming*, 1(6):647–690, 2001.

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

[BSZ14] Filippo Bonchi, Paweł Sobociński, and Fabio Zanasi. A categorical semantics of signal flow graphs. In *International Conference on Concurrency Theory*, pages 435–450. Springer, 2014.

[BSZ15] Filippo Bonchi, Pawel Sobocinski, and Fabio Zanasi. Full abstraction for signal flow graphs. *ACM SIGPLAN Notices*, 50(1):515–526, 2015.

[BSZ17] Filippo Bonchi, Paweł Sobociński, and Fabio Zanasi. Interacting hopf algebras. *Journal of Pure and Applied Algebra*, 221(1):144–184, 2017.

[BSZ18] Filippo Bonchi, Paweł Sobociński, and Fabio Zanasi. Deconstructing lawvere with distributive laws. *Journal of logical and algebraic methods in programming*, 95:128–146, 2018.

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

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

[CGG$^+$22] Geoffrey SH Cruttwell, Bruno Gavranović, Neil Ghani, Paul Wilson, and Fabio Zanasi. Categorical foundations of gradient-based learning. In *European Symposium on Programming*, pages 1–28. Springer, Cham, 2022.

[CK10] Bob Coecke and Aleks Kissinger. The compositional structure of multipartite quantum entanglement. In *International Colloquium on Automata, Languages, and Programming*, pages 297–308. Springer, 2010.

[CK18] Bob Coecke and Aleks Kissinger. Picturing quantum processes. In *International Conference on Theory and Application of Diagrams*, pages 28–31. Springer, 2018.

[CM20] Fabio Gagliardi Cozman and Denis Deratani Mauá. The joy of probabilistic answer set programming: semantics, complexity, expressivity, inference. *International Journal of Approximate Reasoning*, 125:218–239, 2020.

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

[CW87] Aurelio Carboni and Robert FC Walters. Cartesian bicategories i. *Journal of pure and applied algebra*, 49(1-2):11–32, 1987.

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

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

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

[DSBS02] CJ Date, Michael Stonebraker, Paul Brown, and Pedro Soares. Principles of database and knowledge–base systems. *ANALES JAIIO*, 177:188, 2002.

[EGS05] Jason Eisner, Eric Goldlust, and Noah A Smith. Compiling comp ling: Weighted dynamic programming and the dyna language. In *Proceedings of Human Language Technology Conference and Conference on Empirical Methods in Natural Language Processing*, pages 281–290, 2005.

[FJ19] Brendan Fong and Michael Johnson. Lenses and learners. *arXiv preprint arXiv:1903.03671*, 2019.

[Fon13] Brendan Fong. Causal theories: A categorical perspective on bayesian networks. 2013.

[Fox76] Thomas Fox. Coalgebras and cartesian categories. *Communications in Algebra*, 4(7):665–667, 1976.

[Fri20] Tobias Fritz. A synthetic approach to markov kernels, conditional independence and theorems on sufficient statistics. *Advances in Mathematics*, 370:107239, 2020.

[FS19] Brendan Fong and David I Spivak. *An invitation to applied category theory: seven sketches in compositionality.* Cambridge University Press, 2019.

[FSR16] Brendan Fong, Paweł Sobociński, and Paolo Rapisarda. A categorical approach to open and interconnected dynamical systems. In *Proceedings of the 31st annual ACM/IEEE symposium on Logic in Computer Science*, pages 495–504, 2016.

[FST19] Brendan Fong, David Spivak, and Rémy Tuyéras. Backprop as functor: A compositional perspective on supervised learning. In *2019 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–13. IEEE, 2019.

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

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

[Gel87] Michael Gelfond. On stratified autoepistemic theories. In *AAAI*, volume 87, pages 207–211, 1987.

[GL88] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *ICLP/SLP*, volume 88, pages 1070–1080, 1988.

[GPZ22] Tao Gu, Robin Piedeleu, and Fabio Zanasi. A complete diagrammatic calculus for boolean satisfiability. In *38th International Conference on Mathematical Foundations of Programming Semantics (MFPS), Ithaca, USA*, 2022.

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

[GZ21] Tao Gu and Fabio Zanasi. Functorial semantics as a unifying perspective on logic programming. In *Leibniz International Proceedings in Informatics, LIPIcs*, volume 211, page 17. Schloss Dagstuhl, Leibniz-Zentrum, 2021.

[Has97] Masahito Hasegawa. Recursion from cyclic sharing: traced monoidal categories and models of cyclic lambda calculi. In *International Conference on Typed Lambda Calculi and Applications*, pages 196–213. Springer, 1997.

[Has12] Masahito Hasegawa. *Models of Sharing Graphs: A Categorical Semantics of let and letrec.* Springer Science & Business Media, 2012.

[HJS07] Ichiro Hasuo, Bart Jacobs, and Ana Sokolova. Generic trace semantics via coinduction. *arXiv preprint arXiv:0710.2505*, 2007.

[Hun04] Edward V Huntington. Sets of independent postulates for the algebra of logic. *Transactions of the American Mathematical Society*, 5(3):288–309, 1904.

[HV19] Chris Heunen and Jamie Vicary. *Categories for Quantum Theory: an introduction.* Oxford University Press, 2019.

[Jac17] Bart Jacobs. *Introduction to coalgebra*, volume 59. Cambridge University Press, 2017.

[Jac21] Bart Jacobs. From multisets over distributions to distributions over multisets. In *2021 36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–13. IEEE, 2021.

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

[JS91] André Joyal and Ross Street. The geometry of tensor calculus, i. *Advances in mathematics*, 88(1):55–112, 1991.

[JSV96] André Joyal, Ross Street, and Dominic Verity. Traced monoidal categories. In *Mathematical proceedings of the cambridge philosophical society*, volume 119, pages 447–468. Cambridge University Press, 1996.

[JY21] Niles Johnson and Donald Yau. *2-dimensional categories*. Oxford University Press, USA, 2021.

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

[KDDR$^+$10] Angelika Kimmig, Bart Demoen, Luc De Raedt, Vítor Costa, and Ricardo Rocha. On the implementation of the probabilistic logic programming language problog. *Computing Research Repository - CORR*, 11, 06 2010.

[KL80] Gregory M Kelly and Miguel L Laplaza. Coherence for compact closed categories. *Journal of pure and applied algebra*, 19:193–213, 1980.

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

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

[Kli11] Bartek Klin. Bialgebras for structural operational semantics: An introduction. *Theoretical Computer Science*, 412(38):5043–5069, 2011.

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

[Kow88] Robert A Kowalski. The early years of logic programming. *Communications of the ACM*, 31(1):38–43, 1988.

[Koz83] Dexter Kozen. A probabilistic pdl. In *Proceedings of the fifteenth annual ACM symposium on Theory of computing*, pages 291–297, 1983.

[KP96] Yoshiki Kinoshita and A John Power. A fibrational semantics for logic programs. In *International Workshop on Extensions of Logic Programming*, pages 177–191. Springer, 1996.

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

[KP18] Ekaterina Komendantskaya and John Power. Logic programming: Laxness and saturation. *J. Log. Algebr. Meth. Program.*, 101:1–21, 2018.

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

[Lac04] Stephen Lack. Composing props. *Theory and Applications of Categories*, 13(9):147–163, 2004.

[Law63] F William Lawvere. Functorial semantics of algebraic theories. *Proceedings of the National Academy of Sciences*, 50(5):869–872, 1963.

[Law68] F William Lawvere. Some algebraic problems in the context of functorial semantics of algebraic theories. In *Reports of the Midwest Category Seminar II*, pages 41–61. Springer, 1968.

[Lif19] Vladimir Lifschitz. *Answer set programming.* Springer Heidelberg, 2019.

[LKB14] Sławomir Lasota, Bartek Klin, and Mikołaj Bojańczyk. Automata theory in nominal sets. *Logical Methods in Computer Science*, 10, 2014.

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

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

[MDK$^+$18] Robin Manhaeve, Sebastijan Dumancic, Angelika Kimmig, Thomas Demeester, and Luc De Raedt. Deepproblog: Neural probabilistic logic programming. *Advances in Neural Information Processing Systems*, 31, 2018.

[MDR94] Stephen Muggleton and Luc De Raedt. Inductive logic programming: Theory and methods. *The Journal of Logic Programming*, 19:629–679, 1994.

[Mil04] Dale Miller. Overview of linear logic programming. *Linear Logic in Computer Science*, 316:119–150, 2004.

[Mit96] John C Mitchell. *Foundations for programming languages*, volume 1. MIT press Cambridge, 1996.

[MP98] Ugo Montanari and Marco Pistore. An introduction to history dependent automata. *Electronic Notes in Theoretical Computer Science*, 10:170–188, 1998.

[MSB08] 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.

[Pie18] Robin Piedeleu. *Picturing resources in concurrency.* PhD thesis, University of Oxford, 2018.

[Pow16] John Power. Category theoretic semantics for logic programming: Laxness and saturation. *CoALP-Ty'16*, page 3, 2016.

[Prz90] Teodor C. Przymusinski. The well-founded semantics coincides with the three-valued stable semantics. *Fundam. Inform.*, 13:445–463, 1990.

[PZ21] Robin Piedeleu and Fabio Zanasi. A string diagrammatic axiomatisation of finite-state automata. In Stefan Kiefer and Christine Tasson, editors, *Foundations of Software Science and Computation Structures - 24th International Conference, FOSSACS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings*, volume 12650 of *Lecture Notes in Computer Science*, pages 469–489. Springer, 2021.

[Rab63] Michael O Rabin. Probabilistic automata. *Information and control*, 6(3):230–245, 1963.

[RKNP16] Luc De Raedt, Kristian Kersting, Sriraam Natarajan, and David Poole. Statistical relational artificial intelligence: Logic, probability, and computation. *Synthesis lectures on artificial intelligence and machine learning*, 10(2):1–189, 2016.

[Sch61] Marcel Paul Schützenberger. On the definition of a family of automata. *Inf. Control.*, 4(2-3):245–270, 1961.

[Sco70] Dana Scott. *Outline of a mathematical theory of computation.* Oxford University Computing Laboratory, Programming Research Group Oxford, 1970.

[Sel10] Peter Selinger. A survey of graphical languages for monoidal categories. In *New structures for physics*, pages 289–355. Springer, 2010.

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

[SS09] David Speyer and Bernd Sturmfels. Tropical mathematics. *Mathematics Magazine*, 82(3):163–173, 2009.

[Var03] Daniele Varacca. *Probability, nondeterminism and concurrency: two denotational models for probabilistic computation.* BRICS, 2003.

[VGRS91a] 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.

[VGRS91b] Allen Van Gelder, Kenneth A. Ross, and John S. Schlipf. The well-founded semantics for general logic programs. *J. ACM*, 38(3):619–649, jul 1991.

[vHSS17] Gerco van Heerdt, Matteo Sammartino, and Alexandra Silva. Calf: categorical automata learning framework. *arXiv preprint arXiv:1704.05676*, 2017.

[VW06] Daniele Varacca and Glynn Winskel. Distributing probability over nondeterminism. *Mathematical structures in computer science*, 16(1):87–113, 2006.

[WHK10] Günther J Wirsching, Markus Huber, and Christian Kölbl. The confidence-probability semiring. 2010.

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

[XM21] Tom Xu and Yoshihiro Maruyama. Neural string diagrams: A universal modelling language for categorical deep learning. In *International Conference on Artificial General Intelligence*, pages 306–315. Springer, 2021.

[Zan15] Fabio Zanasi. *Interacting Hopf Algebras-the Theory of Linear Systems.* PhD thesis, Ecole normale supérieure de lyon-ENS LYON, 2015.

[ZG21] Fabio Zanasi and Tao Gu. Coalgebraic semantics for probabilistic logic programming. *Logical Methods in Computer Science*, 17, 2021.