BIALGEBRAIC SEMANTICS FOR LOGIC PROGRAMMING

FILIPPO BONCHI AND FABIO ZANASI

Ecole Normale Supérieure de Lyon, CNRS, Inria, UCBL, Université de Lyon — Laboratoire de l'Informatique du Parallélisme.
e-mail address: filippo.bonchi@ens-lyon.fr

Ecole Normale Supérieure de Lyon, CNRS, Inria, UCBL, Université de Lyon — Laboratoire de l'Informatique du Parallélisme.
e-mail address: fabio.zanasi@ens-lyon.fr

Abstract. Bialgebrae provide an abstract framework encompassing the semantics of different kinds of computational models. In this paper we propose a bialgebraic approach to the semantics of logic programming. Our methodology is to study logic programs as reactive systems and exploit abstract techniques developed in that setting. First we use *saturation* to model the operational semantics of logic programs as coalgebrae on presheaves. Then, we make explicit the underlying algebraic structure by using bialgebrae on presheaves. The resulting semantics turns out to be compositional with respect to conjunction and term substitution. Also, it encodes a parallel model of computation, whose soundness is guaranteed by a built-in notion of synchronisation between different threads.

1. Introduction

A fundamental tenet for the semantics of programming languages is *compositionality*: the meaning of a program expression is reducible to the meaning of its subexpressions. This allows inductive reasoning on the structure of programs, and provides several techniques to prove properties of these.

In the last decades, much research has been devoted to develop abstract frameworks for defining compositional semantics for different sorts of computational models. For instance, in the setting of concurrency theory, several rule formats [1] have been introduced for ensuring certain behavioural equivalences to be congruences with respect to the syntactic operators of process algebrae. These works have inspired the Mathematical Operational Semantics by Turi and Plotkin [50] where the semantics of a language is supposed to be a bialgebra for a given distributive law representing a so called *abstract GSOS specification*.

A main drawback for this approach is its poor expressiveness, in the sense that many relevant computational models do not naturally fit into the bialgebraic framework. Prime examples of these, still from concurrency theory, are Petri nets and the calculus of Mobile

1

Ambients [13]: defining the operational behaviour of these systems in terms of their components seems to be an intrinsically complex task that needs serious ingenuity of researchers (see, e.g, [11] for a recent compositional semantics of Petri nets).

Motivated by these considerations, Milner initiated a research program devoted to systematically derive compositional semantics for more flexible semantics specification called *reactive systems* [34]. As shown in [9], these can be modeled as coalgebrae on presheaf categories and the resulting compositional semantics can be obtained by means of *saturation*, a technique that we will detail later.

In this paper we study *logic programs as reactive systems*, applying the aforementioned techniques to obtain a compositional semantics for logic programming. Our approach consists of two steps. First, we model the saturated semantics of a program by means of coalgebrae on presheaves. This allows us to achieve a first form of compositionality, with respect to the substitution of the logic signature. Then, we extend our approach to a bialgebraic setting, making the resulting semantics compositional also with respect to the internal structure of goals.

In the remainder of this introduction, we describe these two steps in more detail.

Coalgebrae on Presheaves and Saturated Semantics. Coalgebrae on presheaves have been successfully employed to provide semantics to *nominal* calculi: sophisticated process calculi with complex mechanisms for variable binding, like the π-calculus [21, 22]. The idea is to have an index category $\mathbf{C}$ of interfaces (or names), and encode as a presheaf $\mathcal{F}\colon \mathbf{C} \to \mathbf{Set}$ the mapping of any object i of $\mathbf{C}$ to the set of states having i as interface, and any arrow $f\colon i \to j$ to a function switching the interface of states from i to j. The operational semantics of the calculus will arise as a notion of transition between states, that is, as a coalgebra $\alpha\colon \mathcal{F} \to \mathcal{B}(\mathcal{F})$, where $\mathcal{B}\colon \mathbf{Set}^{\mathbf{C}} \to \mathbf{Set}^{\mathbf{C}}$ is a functor on presheaves encoding the kind of behavior that we want to express.

As an arrow in a presheaf category, α has to be a natural transformation, i.e. it should commute with arrows $f\colon i \to j$ in the index category $\mathbf{C}$. Unfortunately, this naturality requirement may fail when the structure of $\mathbf{C}$ is rich enough, as for instance when non-injective substitutions [40, 48] or name fusions [39, 6] occur. As a concrete example, consider the π-calculus term $t = \bar{a}\langle x\rangle|b(y)$ consisting of a process $\bar{a}\langle x\rangle$ sending a message x on a channel named a, in parallel with $b(y)$ receiving a message on a channel named b. Since the names a and b are different, the two processes cannot synchronize. Conversely the term $t\theta = \bar{a}\langle x\rangle|a(y)$, that is obtained by applying the substitution θ mapping b to a, can synchronize. If θ is an arrow of the index category $\mathbf{C}$, then the operational semantics α is not natural since $\alpha(t\theta) \neq \alpha(t)\bar{\theta}$, where $\bar{\theta}$ denotes the application of θ to the transitions of t. As a direct consequence, also the unique morphism to the terminal coalgebra is not natural: this means that the abstract semantics of π-calculus is not compositional, in the sense that bisimilarity is not a congruence w.r.t. name substitutions. In order to make bisimilarity a congruence, Sangiorgi introduced in [46] *open bisimilarity*, that is defined by considering the transitions of processes under *all* possible name substitutions θ.

The approach of *saturated semantics* [9] can be seen as a generalization of open bisimilarity, relying on analogous principles: the operational semantics α is "saturated" w.r.t. the arrows of the index category $\mathbf{C}$, resulting in a natural transformation $\alpha^{\sharp}$ in $\mathbf{Set}^{\mathbf{C}}$. In [6, 41], this is achieved by first shifting the definition of α to the category $\mathbf{Set}^{|\mathbf{C}|}$ of presheaves indexed by the discretization $|\mathbf{C}|$ of $\mathbf{C}$. Since $|\mathbf{C}|$ does not have other arrow

than the identities, α is trivially a natural transformation in this setting. The source of α is $\mathcal{U}(\mathcal{F}) \in \mathbf{Set}^{|\mathbf{C}|}$, where $\mathcal{U} \colon \mathbf{Set}^{\mathbf{C}} \to \mathbf{Set}^{|\mathbf{C}|}$ is a forgetful functor defined by composition with the inclusion $\iota \colon |\mathbf{C}| \to \mathbf{C}$. The functor $\mathcal{U}$ has a right adjoint $\mathcal{K} \colon \mathbf{Set}^{|\mathbf{C}|} \to \mathbf{Set}^{\mathbf{C}}$ sending a presheaf to its *right Kan extension* along ι. The adjoint pair $\mathcal{U} \dashv \mathcal{K}$ induces an isomorphism $(\cdot)^{\sharp}_{X,Y} \colon \mathbf{Set}^{|\mathbf{C}|}[\mathcal{U}(X), Y] \to \mathbf{Set}^{\mathbf{C}}[X, \mathcal{K}(Y)]$ mapping α to $\alpha^{\sharp}$. The latter is a natural transformation in $\mathbf{Set}^{\mathbf{C}}$ and, consequently, the abstract semantics results to be compositional.

In the first part of the paper, we show that the saturated approach can be fruitfully instantiated to *coalgebraic logic programming* [30, 32, 31, 33], which consists of a novel semantics for logic programming and a parallel resolution algorithm based on *coinductive trees*. These are a variant of $\wedge\vee$-trees [26] modeling *parallel* implementations of logic programming, where the soundness of the derivations represented by a tree is guaranteed by the restriction to *term-matching* (whose algorithm, differently from unification, is parallelizable [19]).

There are two analogies with the π-calculus: (a) the state space is modeled by a presheaf on the index category $\mathbf{L}^{op}_{\Sigma}$, that is the (opposite) *Lawvere Theory* associated with some signature Σ; (b) the operational semantics given in [32] fails to be a natural transformation in $\mathbf{Set}^{\mathbf{L}^{op}_{\Sigma}}$: Example 3.6 provides a counter-example which is similar to the π-calculus term t discussed above.

The authors of [32] obviate to (b) by relaxing naturality to *lax naturality*: the operational semantics p of a logic program is given as an arrow in the category $Lax(\mathbf{L}^{op}_{\Sigma}, \mathbf{Poset})$ of locally ordered functors $\mathcal{F} \colon \mathbf{L}^{op}_{\Sigma} \to \mathbf{Poset}$ and lax natural transformations between them. They show the existence of a cofree comonad that induces a morphism $\llbracket \cdot \rrbracket_{p}$ mapping atoms (i.e., atomic formulae) to coinductive trees. Since $\llbracket \cdot \rrbracket_{p}$ is not natural but lax natural, the semantics provided by coinductive trees is not compositional, in the sense that, for some atoms A and substitution θ,

$$\llbracket A\theta \rrbracket_{p} \neq \llbracket A \rrbracket_{p}\overline{\theta}$$

where $\llbracket A\theta \rrbracket_{p}$ is the coinductive tree associated with $A\theta$ and $\llbracket A \rrbracket_{p}\overline{\theta}$ denotes the result of applying θ to each atom occurring in the tree $\llbracket A \rrbracket_{p}$.

Instead of introducing laxness, we propose to tackle the non-naturality of p with a saturated approach. It turns out that, in the context of logic programming, the saturation map $(\cdot)^{\sharp}$ has a neat description in terms of substitution mechanisms: while p performs *term-matching* between the atoms and the heads of clauses of a given logic program, its saturation $p^{\sharp}$ (given as a coalgebra in $\mathbf{Set}^{\mathbf{L}^{op}_{\Sigma}}$) performs *unification*. It is worth to remark here that not only most general unifiers are considered but *all* possible unifiers.

A cofree construction leading to a map $\llbracket \cdot \rrbracket_{p^{\sharp}}$ can be obtained by very standard categorical tools, such as terminal sequences [2]. This is possible because, as $\mathbf{Set}$, both $\mathbf{Set}^{\mathbf{L}^{op}_{\Sigma}}$ and $\mathbf{Set}^{|\mathbf{L}^{op}_{\Sigma}|}$ are (co)complete categories, whereas in the lax approach, $Lax(\mathbf{L}^{op}_{\Sigma}, \mathbf{Poset})$ not being (co)complete, more indirect and more sophisticated categorical constructions are needed [31, Sec. 4]. By naturality of $p^{\sharp}$, the semantics given by $\llbracket \cdot \rrbracket_{p^{\sharp}}$ turns out to be compositional, as in the desiderata. Analogously to $\llbracket \cdot \rrbracket_{p}$, also $\llbracket \cdot \rrbracket_{p^{\sharp}}$ maps atoms to tree structures, which we call *saturated $\wedge\vee$-trees*. They generalize coinductive trees, in the sense that the latter can be seen as a "desaturation" of saturated $\wedge\vee$-trees, where all unifiers that are not term-matchers have been discarded. This observation leads to a *translation* from saturated

to coinductive trees, based on the counit ϵ of the adjunction $\mathcal{U} \dashv \mathcal{K}$. It follows that our framework encompasses the semantics in [32, 31].

Analogously to what is done in [31], we propose a notion of *refutation subtree* of a given saturated $\wedge\vee$-tree, intuitively corresponding to an SLD-refutation of an atomic goal in a program. In our approach, not all the refutation subtrees represent sound derivations, because the same variable may be substituted for different terms in the various branches. We thus study the class of *synched* refutation subtrees: they are the ones in which, at each step of the represented derivation, *the same* substitution is applied on all the atoms considered on different branches. Refutation subtrees with this property do represent sound derivations and are preserved by the desaturation procedure. This leads to a result of soundness and completeness of our semantics with respect to SLD-resolution, crucially using both compositionality and the translation into coinductive trees.

Bialgebraic Semantics of Goals. In the second part of this paper we extend our framework to model the saturated semantics of goals instead of single atoms. This broadening of perspective is justified by a second form of compositionality that we want to study. Given atoms A and B, one can see the goal $\{A, B\}$ as their conjunction $A \wedge B$: the idea is that a resolution for $A \wedge B$ requires a resolution for A *and* one for B. Our aim is to take this logical structure into account, proposing a semantics for $A \wedge B$ that can be equivalently given as the "conjunction" (at a higher level) of the semantics $[\![A]\!]_{p^\sharp}$ and $[\![B]\!]_{p^\sharp}$. Formally, we will model the structure given by $\wedge$ as an *algebra* on the space of goals. To properly extend our saturated approach, the coalgebra $p^\sharp$ encoding a logic program needs to be compatible with such algebraic structure: the formal ingredient to achieve this will be a *distributive law* δ involving the type of the algebra of goals and the one of the coalgebra $p^\sharp$. This will give raise to an extension of $p^\sharp$ to a δ-bialgebra $p^\sharp_\delta$ on the space of goals, via a categorical construction that is commonplace in computer science. For instance, when instantiated to a non-deterministic automaton t, it yields the well-known powerset construction $\bar{t}$ on t: the states of t are like atoms of a program, and the ones of $\bar{t}$ are collections of states, like goals are collections of atoms.

Thanks to the compatibility of the operational semantics $p^\sharp_\delta$, the induced map $[\![\cdot]\!]_{p^\sharp_\delta}$ will also be compatible with the algebraic structure of goals, resulting in the compositionality property sketched above,
$$[\![A \wedge B]\!]_{p^\sharp_\delta} = [\![A]\!]_{p^\sharp_\delta} \ \overline{\wedge} \ [\![B]\!]_{p^\sharp_\delta}$$
where on the right side $[\![\cdot]\!]_{p^\sharp_\delta}$ is applied to goals consisting of just one atom, A or B. As we did for $[\![\cdot]\!]_{p^\sharp}$, we will represent the targets of $[\![\cdot]\!]_{p^\sharp_\delta}$ as trees, which we call *saturated $\vee$-trees*. Like saturated $\wedge\vee$-trees, they encode derivations by (generalized) unification, which now apply to goals instead of single atoms. This operational understanding of $[\![\cdot]\!]_{p^\sharp_\delta}$ allows for a more concrete grasp on the higher order conjunction $\overline{\wedge}$: it can be seen as the operation of "gluing together" (depthwise) saturated $\vee$-trees.

Beyond this form of compositionality, our approach exhibits another appealing feature, arising by definition of δ. When solving a goal G in a program $\mathbb{P}$, the operational semantics $p^\sharp_\delta$ attempts to perform unification *simultaneously* on each atom in G with heads in $\mathbb{P}$, by applying *the same* substitution on all atoms in the goal. If this form of "synchronous" resolution is not possible — for instance, if there is no unique substitution for all atoms in G — then the computation encoded by $p^\sharp_\delta$ ends up in a failure.

This behavior is rather different from the one of the standard SLD-resolution algorithm for logic programming, where unification is performed *sequentially* on one atom of the goal at a time. However, we are able to show that the semantics $[\![\cdot]\!]_{p^\sharp_\delta}$ is *sound and complete* with respect to SLD-resolution, meaning that no expressivity is lost in assuming a synchronous derivation procedure as the one above. This result extends the completeness theorem, previously stated for the semantics $[\![\cdot]\!]_{p^\sharp}$, from atomic to arbitrary goals. It relies on a notion of refutation subtree for saturated ∨-trees that is in a sense more natural than the one needed for saturated ∧∨-trees. Whereas in the latter we had to impose specific constraints to ensure that refutation subtrees only represent sound derivations, there is no such need in saturated ∨-trees, because the required synchronisation property is guaranteed by construction.

A Fistful of Trees. The following table summarises how the saturated derivation trees that we introduce compare with the trees appearing in the logic programming literature.

	substitution mechanism		
	most general unification	**term-matching**	**unification**
nodes are atoms	∧∨-trees (Def. 2.6, also called parallel and-or trees [26])	coinductive trees (Def. 3.5, [31, 33])	saturated ∧∨-trees (Def. 4.5)
nodes are goals	SLD-trees (e.g. [35])		saturated ∨-trees (Def. 10.1)

The computation described by SLD-trees is inherently sequential, whereas all the others in the table exhibit and-or parallelism (*cf.* Section 2.5). More specifically, (saturated) ∧∨-trees and coinductive trees express *independent* and-parallelism: there is no exchange of information between the computations involving different atoms in the goal. Instead, saturated ∨-trees encode a dependent form of and-parallelism: at each step *every* atom of the goal is matched with heads of the program, but they have to agree on the same substitution, thus requiring a form of communication between different threads. Since independent and-parallelism does not cohere well with unification, (saturated) ∧∨-trees may represent unsound derivations. Instead, they are always sound by construction in coinductive trees (because of the restriction to term-matching) and saturated ∨-trees (because the atoms in the goal are processed synchronously, by applying the same substitution).

Related works. Our starting point is the key observation that, in coalgebraic logic programming [30, 32, 31, 33], the operational semantics fails to be a natural transformation. As an alternative to the lax approach of the above line of research, we propose saturation which, in the case of logic programming, boils down to unification with respect to all substituions, making the whole approach closer to standard semantics, like Herbrand models [51, 15] (where only ground instances are considered) or the *C*-semantics of [20] (considering also non-ground instances).

As a result, our approach differs sensibly from [30, 32, 31, 33]: in that series of works, the aim is to give an operational semantics to coinductive logic programs and, at the same time, to exploit and-or parallelism. In our case, the semantics is only meant to model standard (recursive) logic programs and the synchronisation mechanism built-in in saturated ∨-trees imposes a form of dependency to and-parallelism.

The two forms of compositionality that we investigate appear in various forms in the standard literature, see e.g. [5, 35]. Our interest is to derive them as the result of applying the categorical machinery — coalgebrae and bialgebrae on presheaves — that has been fruitfully adopted in the setting of process calculi, as detailed above. For instance, in the open π-calculus one aims at the same two forms of compositionality: with respect to name substitution — corresponding to term substitutions in logic programming — and parallel composition of processes — corresponding to conjunction of atoms in a goal.

We should also mention other categorical perspectives on (extensions of) logic programming, such as [18, 28, 4, 9]. Amongst these, the most relevant for us is [9] since it achieves compositionality with respect to substitutions and $\wedge$ by exploiting a form of saturation: arrows of the index category are both substitutions and $\wedge$-contexts of the shape $G_1 \wedge - \wedge G_2$ (for some goals G_1, G_2). The starting observation in [9] is that the construction of relative pushouts [34] instantiated to such category captures the notion of most general unifiers.

Beyond logic programming, the idea of using saturation to achieve compositionality is even older than [46] (see e.g. [42]). As far as we know, [17] is the first work where saturation is explored in terms of coalgebrae. It is interesting to note that, in [16], some of the same authors also proposed laxness as a solution for the lack of compositionality of Petri nets.

A third approach, alternative to laxness and saturation, may be possible by taking a special kind of "powerobject" functor as done in [39, 48] for giving a coalgebraic semantics to fusion and open π-calculus. We have chosen saturated semantics for its generality: it works for any behavioral functor $\mathcal{B}$ and it models a phenomenon that occurs in many different computational models (see e.g. [8]).

Finally, the approach consisting in saturating and building a bialgebraic model already appeared in [23] (amongst others). In that work, a sort of saturated semantics is achieved by transposing along the adjunction between $\mathbf{Set^I}$ and $\mathbf{Set^F}$ obtained from the injection of the category $\mathbf{I}$ of finite sets and injective functions into the category $\mathbf{F}$ of all functions. An interesting construction, missing in [23], is the one of the distributive law δ for saturated semantics: in our work, δ is built in a canonical way out of the distributive law for the non-saturated semantics.

Synopsis. After introducing the necessary background in Section 2, we recall the framework of coalgebraic logic programming of [30, 32, 31, 33] in Section 3. In Section 4 we propose saturated semantics, allowing us to achieve the first compositionality property. In Section 5 we compare saturated semantics with the lax approach of [32]. This is instrumental for proving, in Section 6, soundness and completeness of saturated semantics with respect to SLD-resolution on atomic goals.

In the second part of the paper we present the bialgebraic semantics for arbitrary goals. We start in Section 7 and 8 by considering the simpler setting of ground logic programs. In particular, Section 7 shows the second compositionality property and Section 8 draws a comparison with the coalgebraic semantics. Section 9 and Section 10 generalize the results of the previous two sections to arbitrary logic programs. In particular, we conclude Section 10 by proving soundness and completeness of the bialgebraic semantics of goals with respect to SLD-resolution, extending the analogous result for atomic goals in Section 6.

The present work extends the conference paper [10] with more examples, proofs and the new material presented in Sections 7, 8, 9 and 10.

2. BACKGROUND

In this section we fix some terminology, notation and basic results, mainly concerning category theory and logic programming.

2.1. **Categories and Presheaves.** Given a (small) category $\mathbf{C}$, $|\mathbf{C}|$ denotes the category with the same objects as $\mathbf{C}$ but no other arrow than the identities. With a little abuse of notation, $X \in \mathbf{C}$ indicates that X is an object of $\mathbf{C}$ and $\mathbf{C}[X,Y]$ the set of arrows from X to Y. We denote with $X \times Y$ the product of objects $X, Y \in \mathbf{C}$ with projections $\pi_1 \colon X \times Y \to X$ and $\pi_2 \colon X \times Y \to Y$. Given $Z \in \mathbf{C}$ and arrows $f \colon Z \to X$ and $g \colon Z \to Y$, we denote with $\langle f, g \rangle \colon Z \to X \times Y$ the arrow given by universal property of $X \times Y$. We use the notation $\mathbf{End}(\mathbf{C})$ for the category of endofunctors on $\mathbf{C}$ and natural transformations. A $\mathbf{C}$-indexed *presheaf* is any functor $\mathcal{G} \colon \mathbf{C} \to \mathbf{Set}$. We write $\mathbf{Set}^{\mathbf{C}}$ for the category of $\mathbf{C}$-indexed presheaves and natural transformations.

Throughout this paper we will need to extend functors on $\mathbf{Set}$ to functors on presheaf categories. For this purpose, it will be useful to state the following construction.

Definition 2.1. Given a category $\mathbf{C}$, the functor $(\cdot)^{\mathbf{C}} \colon \mathbf{End}(\mathbf{Set}) \to \mathbf{End}(\mathbf{Set}^{\mathbf{C}})$ is defined as follows.

- Given an object $\mathcal{F} \colon \mathbf{Set} \to \mathbf{Set}$, $\mathcal{F}^{\mathbf{C}} \colon \mathbf{Set}^{\mathbf{C}} \to \mathbf{Set}^{\mathbf{C}}$ is defined on $\mathcal{G} \colon \mathbf{C} \to \mathbf{Set}$ as $\mathcal{F} \circ \mathcal{G}$ and on $\alpha \colon \mathcal{G} \Rightarrow \mathcal{H}$ as the natural transformation given by $\left(\mathcal{F}\mathcal{G}(n) \xrightarrow{\mathcal{F}(\alpha_n)} \mathcal{F}\mathcal{H}(n) \right)_{n \in \mathbf{C}}$.
- Given an arrow $\gamma \colon \mathcal{F} \Rightarrow \mathcal{B}$ of $\mathbf{End}(\mathbf{Set})$, $\gamma^{\mathbf{C}} \colon \mathcal{F}^{\mathbf{C}} \Rightarrow \mathcal{B}^{\mathbf{C}}$ is a natural transformation β given by the family $\left(\mathcal{F}\mathcal{G} \xrightarrow{\beta_{\mathcal{G}}} \mathcal{B}\mathcal{G} \right)_{\mathcal{G} \in \mathbf{Set}^{\mathbf{C}}}$ of natural transformations, where each $\beta_{\mathcal{G}}$ is defined by $\left(\mathcal{F}\mathcal{G}(n) \xrightarrow{\gamma_{\mathcal{G}(n)}} \mathcal{B}\mathcal{G}(n) \right)_{n \in \mathbf{C}}$.

We call $\mathcal{F}^{\mathbf{C}}$ and $\gamma^{\mathbf{C}}$ *extensions* of $\mathcal{F}$ and γ respectively.

We will mainly work with categories of presheaves indexed by $\mathbf{L}_{\mathbf{\Sigma}}^{op}$ — the (opposite) Lawvere theory on a signature Σ, defined in Section 2.4 — and its discretization $|\mathbf{L}_{\mathbf{\Sigma}}^{op}|$. To simplify notation, we will follow the convention of writing $\check{(\cdot)}$ for the extension functor $(\cdot)^{\mathbf{L}_{\mathbf{\Sigma}}^{op}}$, i.e., $(\cdot)^{\mathbf{C}}$ where $\mathbf{C} = \mathbf{L}_{\mathbf{\Sigma}}^{op}$, and $\widehat{(\cdot)}$ for $(\cdot)^{|\mathbf{L}_{\mathbf{\Sigma}}^{op}|}$.

2.2. **Monads and Distributive Laws.** A *monad* in a category $\mathbf{C}$ is a functor $\mathcal{T} \colon \mathbf{C} \to \mathbf{C}$ together with two natural transformations $\eta \colon id \Rightarrow \mathcal{T}$ and $\mu \colon \mathcal{T}\mathcal{T} \Rightarrow \mathcal{T}$, called respectively unit and multiplication of $\mathcal{T}$, which are required to satisfy the following equations for any $X \in \mathbf{C}$: $\mu_X \circ \eta_{\mathcal{T}X} = \mu_X \circ \mathcal{T}\eta_X = id_X$ and $\mathcal{T}\mu_X \circ \mu_{XX} = \mu_X \circ \mu_X$. We shall also make use of the triple notation $(\mathcal{T}, \eta, \mu)$ for monads. A distributive law of a *monad* $(\mathcal{T}, \eta^{\mathcal{T}}, \mu^{\mathcal{T}})$ over a

monad $(\mathcal{M}, \eta^{\mathcal{M}}, \mu^{\mathcal{M}})$ is a natural transformation $\lambda\colon \mathcal{T}\mathcal{M} \Rightarrow \mathcal{M}\mathcal{T}$ making the diagrams below commute:

$$
\begin{array}{ccc}
\mathcal{T}X \,=\!=\!=\, \mathcal{T}X & \quad & \mathcal{T}\mathcal{M}^2 X \xrightarrow{\lambda_{\mathcal{M}X}} \mathcal{M}\mathcal{T}\mathcal{M}X \xrightarrow{\mathcal{M}\lambda_X} \mathcal{M}^2\mathcal{T}X
\end{array}
$$

A distributive law $\lambda\colon \mathcal{T}\mathcal{M} \Rightarrow \mathcal{M}\mathcal{T}$ between monads yields a monad $\mathcal{M}\mathcal{T}$ with unit $\eta^{\mathcal{M}\mathcal{T}} :=$ $\eta^{\mathcal{M}}_{\mathcal{T}} \circ \eta^{\mathcal{T}}$ and multiplication $\mu^{\mathcal{M}\mathcal{T}} := \mathcal{M}\mu^{\mathcal{T}} \circ \mu^{\mathcal{M}}_{\mathcal{T}\mathcal{T}} \circ \mathcal{M}\lambda_{\mathcal{T}}$. We introduce now two weaker notions of the law. A distributive law of a *monad* $(\mathcal{T}, \eta^{\mathcal{T}}, \mu^{\mathcal{T}})$ over a *functor* $\mathcal{M}$ is a natural transformation $\lambda\colon \mathcal{T}\mathcal{M} \Rightarrow \mathcal{M}\mathcal{T}$ such that only the two bottommost squares above commute. One step further in generalization, we call a distributive law of a *functor* $\mathcal{T}$ over a *functor* $\mathcal{M}$ any natural transformation from $\mathcal{T}\mathcal{M}$ to $\mathcal{M}\mathcal{T}$.

With the next proposition we observe that the extension functor $(\cdot)^{\mathbf{C}}$ (Definition 2.1) preserves the monad structure.

Proposition 2.2. *If $(\mathcal{T}, \eta, \mu)$ is a monad in $\mathbf{Set}$ then $(\mathcal{T}^{\mathbf{C}}, \eta^{\mathbf{C}}, \mu^{\mathbf{C}})$ is a monad in $\mathbf{Set}^{\mathbf{C}}$. Moreover, if $\lambda\colon \mathcal{T}\mathcal{M} \to \mathcal{M}\mathcal{T}$ is a distributive law of monads in $\mathbf{Set}$ then $\lambda^{\mathbf{C}}\colon \mathcal{T}^{\mathbf{C}}\mathcal{M}^{\mathbf{C}} \to \mathcal{M}^{\mathbf{C}}\mathcal{T}^{\mathbf{C}}$ is a distributive law of monads in $\mathbf{Set}^{\mathbf{C}}$.*

Proof. The action of $(\cdot)^{\mathbf{C}}$ on arrows of $\mathbf{End}(\mathbf{Set})$ is given componentwise. This means that commutativity of the diagrams involving $\eta^{\mathbf{C}}$, $\mu^{\mathbf{C}}$ and $\lambda^{\mathbf{C}}$ in $\mathbf{Set}^{\mathbf{C}}$ follows by the one of the corresponding diagrams in $\mathbf{Set}$. $\qquad\square$

2.3. **Algebrae, Coalgebrae and Bialgebrae.** Given a functor $\mathcal{F}\colon \mathbf{C} \to \mathbf{C}$, a $\mathcal{F}$-*algebra* on $X \in \mathbf{C}$ is an arrow $h\colon \mathcal{F}(X) \to X$, also written as a pair (X, h). A morphism between $\mathcal{F}$-algebrae (X, h) and (Y, i) is an arrow $f\colon X \to Y$ such that $f \circ h = i \circ \mathcal{F}(f)$.

Dually, a $\mathcal{B}$-*coalgebra* on $X \in \mathbf{C}$ is an arrow $p\colon X \to \mathcal{B}(X)$, also written as a pair (X, p). A morphism between $\mathcal{B}$-coalgebrae (X, p) and (Y, q) is an arrow $g\colon X \to Y$ such that $q \circ g = \mathcal{B}(g) \circ p$. We fix notation $\mathbf{CoAlg}(\mathcal{B})$ for the category of $\mathcal{B}$-coalgebrae and their morphisms. If it exists, the *final $\mathcal{B}$-coalgebra* is the terminal object in $\mathbf{CoAlg}(\mathcal{B})$. The *cofree $\mathcal{B}$-coalgebra* on $X \in \mathbf{C}$ is given by $(\Omega, \pi_2 \circ \omega\colon \Omega \to \mathcal{B}(\Omega))$, where (Ω, ω) is the terminal object in $\mathbf{CoAlg}(X \times \mathcal{B}(\cdot))$.

Let $\lambda\colon \mathcal{F}\mathcal{B} \Rightarrow \mathcal{B}\mathcal{F}$ be a distributive law between functors $\mathcal{F}, \mathcal{B}\colon \mathbf{C} \to \mathbf{C}$. A λ-*bialgebra* is a triple (X, h, p) where $h\colon \mathcal{F}X \to X$ is an $\mathcal{F}$-algebra and $p\colon X \to \mathcal{B}X$ is a $\mathcal{B}$-coalgebra subject to the compatibility property given by commutativity of the following diagram:

$$
\begin{array}{ccc}
\mathcal{F}X & \xrightarrow{\ h\ } X \xrightarrow{\ p\ } & \mathcal{B}X \\
{\scriptstyle \mathcal{F}p}\big\downarrow & & \big\uparrow{\scriptstyle \mathcal{B}h} \\
\mathcal{F}\mathcal{B}X & \xrightarrow[\ \lambda_X\ ]{} & \mathcal{B}\mathcal{F}X
\end{array}
$$

A λ-*bialgebra morphism* from (X, h, p) to (Y, i, q) is an arrow $f\colon X \to Y$ that is both a $\mathcal{B}$-coalgebra morphism from (X, p) to (Y, q) and an $\mathcal{F}$-algebra morphism from (X, h) to

(Y, i). We fix notation $\mathbf{BiAlg}(\lambda)$ for the category of λ-bialgebrae and their morphisms. If it exists, the *final λ-bialgebra* is the terminal object in $\mathbf{BiAlg}(\lambda)$.

Next we record some useful constructions of bialgebrae out of coalgebrae. When $\mathcal{F}$ is a monad and λ is a distributive law of the *monad* $\mathcal{F}$ over the functor $\mathcal{B}$, then any $\mathcal{BF}$-coalgebra canonically lifts to a λ-bialgebra as guaranteed by the following proposition (for a proof see e.g. [27]).

Proposition 2.3. *Given a monad $\mathcal{T}\colon \mathbf{C} \to \mathbf{C}$ and a functor $\mathcal{B}\colon \mathbf{C} \to \mathbf{C}$, let $\lambda\colon \mathcal{TB} \Rightarrow \mathcal{BT}$ be a distributive law of the monad $\mathcal{T}$ over the functor $\mathcal{B}$. Given a $\mathcal{BT}$-coalgebra $p\colon X \to \mathcal{BT}X$, define the $\mathcal{B}$-coalgebra $p_\lambda\colon \mathcal{T}X \to \mathcal{BT}X$ as*

$$\mathcal{T}X \xrightarrow{\ \mathcal{T}p\ } \mathcal{TBT}X \xrightarrow{\ \lambda_{\mathcal{T}X}\ } \mathcal{BTT}X \xrightarrow{\ \mathcal{B}(\mu_X^{\mathcal{T}})\ } \mathcal{BT}X \ .$$

Then $(\mathcal{T}X, \mu_X^{\mathcal{T}}, p_\lambda)$ forms a λ-bialgebra. Moreover, this assignment extends to a functor from $\mathbf{CoAlg}(\mathcal{BT})$ to $\mathbf{BiAlg}(\lambda)$ mapping:

- *a $\mathcal{BT}$-coalgebra (X, p) to the λ-bialgebra $(\mathcal{T}X, \mu_X^{\mathcal{T}}, p_\lambda)$;*
- *a morphism $f\colon X \to Y$ of $\mathcal{BT}$-coalgebrae to a morphism $\mathcal{T}f\colon \mathcal{T}X \to \mathcal{T}Y$ of λ-bialgebrae.*

We recall also the following folklore result (see e.g. [29]) on the relation between final coalgebrae and final bialgebrae.

Proposition 2.4. *Let $\lambda\colon \mathcal{FB} \Rightarrow \mathcal{BF}$ be a distributive law between functors $\mathcal{F}, \mathcal{B}\colon \mathbf{C} \to \mathbf{C}$ and suppose that a final $\mathcal{B}$-coalgebra $c\colon X \to \mathcal{B}X$ exists. Form the $\mathcal{B}$-coalgebra $h\colon \mathcal{F}X \xrightarrow{\mathcal{F}c} \mathcal{FB}X \xrightarrow{\lambda_X} \mathcal{BF}X$ and let $h'\colon \mathcal{F}X \to X$ be the unique $\mathcal{B}$-coalgebra morphism given by finality of $c\colon X \to \mathcal{B}X$. Then (X, h', c) is the final λ-bialgebra.*

2.4. **Terms, Atoms and Substitutions.** We fix a *signature* Σ of function symbols, each equipped with a fixed arity, and a countably infinite set $Var = \{x_1, x_2, x_3, \dots\}$ of variables. We model substitutions and unification of terms over Σ and Var according to the categorical perspective of [24, 12]. To this aim, let the (opposite) *Lawvere Theory* of Σ be a category $\mathbf{L}_{\Sigma}^{op}$ where objects are natural numbers, with $n \in \mathbf{L}_{\Sigma}^{op}$ intuitively representing variables $x_1, x_2, \dots, x_n$ from Var. For any two $n, m \in \mathbf{L}_{\Sigma}^{op}$, the set $\mathbf{L}_{\Sigma}^{op}[n, m]$ consists of all n-tuples $\langle t_1, \dots, t_n \rangle$ of terms where only variables among $x_1, \dots, x_m$ occur. The identity on $n \in \mathbf{L}_{\Sigma}^{op}$, denoted by id_n, is given by the tuple $\langle x_1, \dots, x_n \rangle$. The composition of $\langle t_1^1, \dots, t_n^1 \rangle\colon n \to m$ and $\langle t_1^2, \dots, t_m^2 \rangle\colon m \to m'$ is the tuple $\langle t_1, \dots, t_n \rangle\colon n \to m'$, where t_i is the term t_i^1 in which every variable x_j has been replaced with t_j^2, for $1 \leq j \leq m$ and $1 \leq i \leq n$.

We call *substitutions* the arrows of $\mathbf{L}_{\Sigma}^{op}$ and use Greek letters θ, σ and τ to denote them. Given $\theta_1\colon n \to m_1$ and $\theta_2\colon n \to m_2$, a *unifier* of θ_1 and θ_2 is a pair of substitutions $\sigma\colon m_1 \to m$ and $\tau\colon m_2 \to m$, where m is some object of $\mathbf{L}_{\Sigma}^{op}$, such that $\sigma \circ \theta_1 = \tau \circ \theta_2$. The *most general unifier* of θ_1 and θ_2 is a unifier with a universal property, i.e. a pushout of the diagram $m_1 \xleftarrow{\theta_1} n \xrightarrow{\theta_2} m_2$.

An *alphabet* $\mathcal{A}$ consists of a signature Σ, a set of variables Var and a set of predicate symbols $P, P_1, P_2, \dots$ each assigned an arity. Given P of arity n and Σ-terms $t_1, \dots, t_n$, $P(t_1, \dots, t_n)$ is called an *atom*. We use Latin capital letters $A, B, \dots$ for atoms. Given a substitution $\theta = \langle t_1, \dots, t_n \rangle\colon n \to m$ and an atom A with variables among $x_1, \dots, x_n$, we adopt the standard notation of logic programming in denoting with $A\theta$ the atom obtained

by replacing x_i with t_i in A, for $1 \leq i \leq n$. The atom $A\theta$ is called a *substitution instance* of A. The notation $\{A_1, \ldots, A_m\}\theta$ is a shorthand for $\{A_1\theta, \ldots, A_m\theta\}$. Given atoms A_1 and A_2, we say that A_1 *unifies* with A_2 (equivalently, they are *unifiable*) if they are of the form $A_1 = P(t_1, \ldots, t_n)$, $A_2 = P(t'_1, \ldots, t'_n)$ and a unifier $\langle \sigma, \tau \rangle$ of $\langle t_1, \ldots, t_n \rangle$ and $\langle t'_1, \ldots, t'_n \rangle$ exists. Observe that, by definition of unifier, this amounts to saying that $A_1\sigma = A_2\tau$. *Term matching* is a particular case of unification, where σ is the identity substitution. In this case we say that $\langle \sigma, \tau \rangle$ is a *term-matcher* of A_1 and A_2, meaning that $A_1 = A_2\tau$.

2.5. **Logic Programming.** A *logic program* $\mathbb{P}$ consists of a finite set of *clauses* C written as $H \leftarrow B_1, \ldots, B_k$. The components H and $B_1, \ldots, B_k$ are atoms, where H is called the *head* of C and $B_1, \ldots, B_k$ form the *body* of C. One can think of $H \leftarrow B_1, \ldots, B_k$ as representing the first-order formula $(B_1 \wedge \cdots \wedge B_k) \to H$. We say that $\mathbb{P}$ is *ground* if only ground atoms (i.e. without variables) occur in its clauses.

The central algorithm of logic programming is SLD-resolution, checking whether a finite collection of atoms G (called a *goal* and usually modeled as a set or a list) is *refutable* in $\mathbb{P}$. A run of the algorithm on inputs G and $\mathbb{P}$ gives rise to an *SLD-derivation*, whose steps of computation can be sketched as follows. At the initial step 0, a set of atoms G_0 (the current goal) is initialized as G. At each step i, an atom A_i is selected in the current goal G_i and one checks whether A_i is unifiable with the head of some clause of the program. If not, the computation terminates with a failure. Otherwise, one such clause $C_i = H \leftarrow B_1, \ldots, B_k$ is selected: by a classical result, since A_i and H unify, they have also a most general unifier $\langle \sigma_i, \tau_i \rangle$. The goal G_{i+1} for the step $i+1$ is given as

$$\{B_1, \ldots, B_k\}\tau_i \ \cup \ (G_i \setminus \{A_i\})\sigma_i.$$

Such a computation is called an *SLD-refutation* if it terminates in a finite number (say n) of steps with $G_n = \emptyset$. In this case one calls *computed answer* the substitution given by composing the first projections $\sigma_n, \ldots, \sigma_0$ of the most general unifiers associated with each step of the computation. The goal G is *refutable* in $\mathbb{P}$ if an SLD-refutation for G in $\mathbb{P}$ exists. A *correct answer* is a substitution θ for which there exist a computed answer τ and a substituion σ such that $\sigma \circ \tau = \theta$. We refer to [35] for a more detailed introduction to SLD-resolution.

Convention 2.5. In any derivation of G in $\mathbb{P}$, the standard convention is that the variables occurring in the clause C_i considered at step i do not appear in goals $G_{i-1}, \ldots, G_0$. This guarantees that the computed answer is a well-defined substitution and may require a dynamic (i.e. at step i) renaming of variables appearing in C_i. The associated procedure is called *standardizing the variables apart* and we assume it throughout the paper without explicit mention. It also justifies our definition (Section 2.4) of the most general unifier as pushout of two substitutions *with different target*, whereas it is also modeled in the literature as the coequalizer of two substitutions *with the same target*, see e.g. [24]. The different target corresponds to the two substitutions depending on disjoint sets of variables.

Relevant for our exposition are $\wedge\vee$-*trees* ("and-or trees") [26], which represent executions of SLD-resolution exploiting two forms of parallelism: *and-parallelism*, corresponding to simultaneous refutation-search of multiple atoms in a goal, and *or-parallelism*, exploring multiple attempts to refute the same goal. These are also called *and-or parallel derivation tress* in [33].

Definition 2.6. Given a logic program $\mathbb{P}$ and an atom A, the (parallel) $\wedge\vee$-*tree* for A in $\mathbb{P}$ is the possibly infinite tree T satisfying the following properties:

(1) Each node in T is either an $\wedge$-node or an $\vee$-node.
(2) Each $\wedge$-node is labeled with one atom and its children are $\vee$-nodes.
(3) The root of T is an $\wedge$-node labeled with A.
(4) Each $\vee$-node is labeled with $\bullet$ and its children are $\wedge$-nodes.
(5) For every $\wedge$-node s in T, let A' be its label. For every clause $H \leftarrow B_1, \ldots, B_k$ of $\mathbb{P}$ and most general unifier $\langle \sigma, \tau \rangle$ of A' and H, s has exactly one child t, and viceversa. For each atom B in $\{B_1, \ldots, B_k\}\tau$, t has exactly one child labeled with B, and viceversa.

As standard for any tree, we have a notion of *depth*: the root is at depth 0 and depth $i+1$ is given by the children of nodes at depth i. In our graphical representation of trees, we draw arcs between a node and its children which we call *edges*. A *subtree* of a tree T is a set of nodes of T forming a tree T', such that the child relation between nodes in T' agrees with the one they have in T.

An SLD-resolution for the singleton goal $\{A\}$ is represented as a particular kind of subtree of the $\wedge\vee$-tree for A, called *derivation subtree*. The intuition is that derivation subtrees encode "deterministic" computations, that is, no branching given by or-parallelism is allowed. *Refutation subtrees* are those derivation subtrees yielding an SLD-refutation of $\{A\}$: all paths lead to a leaf with no atoms left to be refuted.

Definition 2.7. Let T be the $\wedge\vee$-tree for an atom A in a program $\mathbb{P}$. A subtree T' of T is a *derivation subtree* if it satisfies the following conditions:

(1) the root of T' is the root of T;
(2) if an $\wedge$-node of T belongs to T', then just one of its children belongs to T';
(3) if an $\vee$-node of T belongs to T', then all its children belong to T'.

A *refutation subtree* (called success subtree in [31]) is a finite derivation subtree with only $\vee$-nodes as leaves.

3. Coalgebraic Logic Programming

In this section we recall the framework of coalgebraic logic programming, as introduced in [30, 32, 31, 33].

3.1. **The Ground Case.** We begin by considering the coalgebraic semantics of ground logic programs [30]. For the sequel we fix an alphabet $\mathcal{A}$, a set At of ground atoms and a ground logic program $\mathbb{P}$. The behavior of $\mathbb{P}$ is represented by a coalgebra $p \colon At \to \mathcal{P}_f\mathcal{P}_f(At)$ on **Set**, where $\mathcal{P}_f$ is the finite powerset functor and p is defined as follows:

$$p \colon A \ \mapsto \ \{\{B_1, \ldots, B_k\} \ \mid \ H \leftarrow B_1, \ldots, B_k \text{ is a clause of } \mathbb{P} \text{ and } A = H\}.$$

The idea is that p maps an atom $A \in At$ to the set of bodies of clauses of $\mathbb{P}$ whose head H unifies with A, i.e. (in the ground case) $A = H$. Therefore $p(A) \in \mathcal{P}_f\mathcal{P}_f(At)$ can be seen as representing the $\wedge\vee$-tree of A in $\mathbb{P}$ up to depth 2, according to Definition 2.6: each element $\{B_1, \ldots, B_k\}$ of $p(A)$ corresponds to a child of the root, whose children are labeled with $B_1, \ldots, B_k$. The full tree is recovered as an element of $\mathcal{C}(\mathcal{P}_f\mathcal{P}_f)(At)$, where $\mathcal{C}(\mathcal{P}_f\mathcal{P}_f)$ is the *cofree comonad* on $\mathcal{P}_f\mathcal{P}_f$, standardly provided by the following construction [2, 52].

Construction 3.1. The terminal sequence for the functor $At \times \mathcal{P}_f \mathcal{P}_f(\cdot) \colon \mathbf{Set} \to \mathbf{Set}$ consists of sequences of objects X_α and arrows $\zeta_\alpha \colon X_{\alpha+1} \to X_\alpha$, defined by induction on α as follows.

$$X_\alpha \coloneqq \begin{cases} At & \alpha = 0 \\ At \times \mathcal{P}_f \mathcal{P}_f(X_\beta) & \alpha = \beta + 1 \end{cases} \qquad \zeta_\alpha \coloneqq \begin{cases} \pi_1 & \alpha = 0 \\ id_{At} \times \mathcal{P}_f \mathcal{P}_f(\zeta_\beta) & \alpha = \beta + 1 \end{cases}$$

For α a limit ordinal, X_α is given as a limit of the sequence and a function $\zeta_\alpha \colon X_\alpha \to X_\beta$ is given for each $\beta < \alpha$ by the limiting property of X_α.

By [52] it follows that the sequence given above converges to a limit X_γ such that $\zeta_\gamma \colon X_{\gamma+1} \to X_\gamma$ is an isomorphism forming the final $At \times \mathcal{P}_f \mathcal{P}_f(\cdot)$-coalgebra $(X_\gamma, \zeta_\gamma^{-1})$. Since $X_{\gamma+1}$ is defined as $At \times \mathcal{P}_f \mathcal{P}_f(X_\gamma)$, there is a projection function $\pi_2 \colon X_{\gamma+1} \to \mathcal{P}_f \mathcal{P}_f(X_\gamma)$ which makes $\pi_2 \circ \zeta_\gamma^{-1} \colon X_\gamma \to \mathcal{P}_f \mathcal{P}_f(X_\gamma)$ the *cofree $\mathcal{P}_f \mathcal{P}_f$-coalgebra* on At. This induces the *cofree comonad* $\mathcal{C}(\mathcal{P}_f \mathcal{P}_f) \colon \mathbf{Set} \to \mathbf{Set}$ on $\mathcal{P}_f \mathcal{P}_f$ as a functor mapping At to X_γ.

As the elements of the cofree $\mathcal{P}_f$-coalgebra on a set X are standardly presented as finitely branching trees where nodes have elements of X as labels [52, 45], those of the cofree $\mathcal{P}_f \mathcal{P}_f$-coalgebra on X can be seen as finitely branching trees with two sorts of nodes occurring at alternating depth, where only one sort has the X-labeling. We now define a $\mathcal{C}(\mathcal{P}_f \mathcal{P}_f)$-coalgebra $[\![\cdot]\!]_p \colon At \to \mathcal{C}(\mathcal{P}_f \mathcal{P}_f)(At)$.

Construction 3.2. Given a ground program $\mathbb{P}$, let $p \colon At \to \mathcal{P}_f \mathcal{P}_f(At)$ be the coalgebra associated with $\mathbb{P}$. We define a cone $\{p_\alpha \colon At \to X_\alpha\}_{\alpha < \gamma}$ on the terminal sequence of Construction 3.1 as follows:

$$p_\alpha \quad \coloneqq \quad \begin{cases} id_{At} & \alpha = 0 \\ \langle id_{At}, (\mathcal{P}_f \mathcal{P}_f(p_\beta) \circ p) \rangle & \alpha = \beta + 1. \end{cases}$$

For α a limit ordinal, $p_\alpha \colon At \to X_\alpha$ is provided by the limiting property of X_α. Then in particular $X_\gamma = \mathcal{C}(\mathcal{P}_f \mathcal{P}_f)(At)$ yields a function $[\![\cdot]\!]_p \colon At \to \mathcal{C}(\mathcal{P}_f \mathcal{P}_f)(At)$.

Given an atom $A \in At$, the tree $[\![A]\!]_p \in \mathcal{C}(\mathcal{P}_f \mathcal{P}_f)(At)$ is built by iteratively applying the map p, first to A, then to each atom in $p(A)$, and so on. For each natural number m, p_m maps A to its $\wedge\vee$-tree up to depth m. As shown in [30], the limit $[\![\cdot]\!]_p$ of all such approximations provides the full $\wedge\vee$-tree of A.

Example 3.3. Consider the following ground logic program, based on an alphabet consisting of a signature $\{a^0, b^0, c^0\}$ and predicates $\mathsf{p}(-, -)$, $\mathsf{q}(-)$.

$$\mathsf{p}(b, c) \ \leftarrow \ \mathsf{q}(a), \mathsf{q}(b), \mathsf{q}(c) \qquad\qquad \mathsf{p}(b, b) \ \leftarrow \ \mathsf{q}(c)$$
$$\mathsf{p}(b, b) \ \leftarrow \ \mathsf{p}(b, a), \mathsf{p}(b, c) \qquad\qquad \mathsf{q}(c) \ \leftarrow$$

The corresponding coalgebra $p \colon At \to \mathcal{P}_f \mathcal{P}_f(At)$ and the $\wedge\vee$-tree $[\![\mathsf{p}(b,b)]\!]_p \in \mathcal{C}(\mathcal{P}_f \mathcal{P}_f)(At)$ are depicted below on the left and on the right, respectively.

$$\begin{aligned} p(\mathsf{p}(b, c)) &= \{\{\mathsf{q}(a), \mathsf{q}(b), \mathsf{q}(c)\}\} \\ p(\mathsf{p}(b, b)) &= \{\{\mathsf{p}(b, a), \mathsf{p}(b, c)\}\{\mathsf{q}(c)\}\} \\ p(\mathsf{q}(c)) &= \{\{\}\} \\ p(A) &= \{\} \text{ for } A \in At \setminus \{\mathsf{p}(b, c), \mathsf{p}(b, b), \mathsf{q}(c)\} \end{aligned}$$
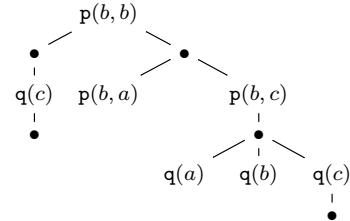
3.2. **The General Case.** In the sequel we recall the extension of the coalgebraic semantics to arbitrary (i.e. possibly non-ground) logic programs presented in [32, 31]. A motivating observation for their approach is that, in presence of variables, $\wedge\vee$-trees are not guaranteed to represent sound derivations. The problem lies in the interplay between variable dependencies and unification, which makes SLD-derivations for logic programs inherently *sequential* processes [19].

Example 3.4. Consider the signature $\Sigma = \{cons^2, succ^1, zero^0, nil^0\}$ and the predicates $\mathtt{List}(-)$, $\mathtt{Nat}(-)$. The program $\mathtt{NatList}$, encoding the definition of lists of natural numbers, will be our running example of a non-ground logic program.

$$\mathtt{List}(cons(x_1, x_2)) \leftarrow \mathtt{Nat}(x_1), \mathtt{List}(x_2) \qquad\qquad \mathtt{List}(nil) \leftarrow$$
$$\mathtt{Nat}(succ(x_1)) \leftarrow \mathtt{Nat}(x_1) \qquad\qquad\qquad \mathtt{Nat}(zero) \leftarrow$$

Let A be the atom $\mathtt{List}(cons(x_1, cons(x_2, x_1)))$. It is intuitively clear that there is no substitution of variables making A represent a list of natural numbers: we should replace x_1 with a "number" (for instance $zero$) in its first occurrence and with a "list" (for instance nil) in its second occurrence. Consequently, there is no SLD-refutation for $\{A\}$ in $\mathtt{NatList}$. However, consider the $\wedge\vee$-tree of A in $\mathtt{NatList}$, for which we provide a partial representation as follows.
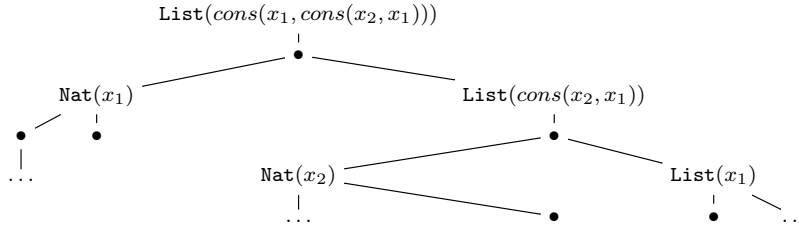
The above tree seems to yield an SLD-refutation: $\mathtt{List}(cons(x_1, cons(x_2, x_1)))$ is refuted by proving $\mathtt{Nat}(x_1)$ and $\mathtt{List}(cons(x_2, x_1))$. However, the associated computed answer would be ill-defined, as it is given by substituting x_2 with $zero$ and x_1 both with $zero$ and with nil (the computed answer of $\mathtt{Nat}(x_1)$ maps x_1 to $zero$ and the computed answer of $\mathtt{List}(cons(x_2, x_1))$ maps x_1 to nil).

To obviate to this problem, in [32] *coinductive trees* are introduced as a sound variant of $\wedge\vee$-trees, where unification is restricted to term-matching. This constraint is sufficient to guarantee that coinductive trees only represent sound derivations: the key intuition is that a term-matcher is a unifier that leaves untouched the current goal, meaning that the "previous history" of the derivation remains uncorrupted.

Before formally defining coinductive trees, it is worth recalling that, in [32], the collection of atoms (based on an alphabet $\mathcal{A}$) is modeled as a presheaf $At \colon \mathbf{L}_\Sigma^{op} \to \mathbf{Set}$. The index category is the (opposite) *Lawvere Theory* $\mathbf{L}_\Sigma^{op}$ of Σ, as defined above. For each natural number $n \in \mathbf{L}_\Sigma^{op}$, $At(n)$ is defined as the set of atoms with variables among $x_1, \ldots, x_n$. Given an arrow $\theta \in \mathbf{L}_\Sigma^{op}[n, m]$, the function $At(\theta) \colon At(n) \to At(m)$ is defined by substitution, i.e. $At(\theta)(A) := A\theta$. By definition, whenever an atom A belongs to $At(n)$, then it also belongs to $At(n')$, for all $n' \geq n$. However, the occurrences of the same atom in $At(n)$ and $At(n')$ (for $n \neq n'$) are considered distinct: the atoms $A \in At(n)$ and $A \in At(n')$ can be thought of as two states $x_1, \ldots, x_n \vdash A$ and $x_1, \ldots, x_{n'} \vdash A$ with two different interfaces $x_1, \ldots, x_n$ and $x_1, \ldots, x_{n'}$. For this reason, when referring to an atom A, it is important to always specify the set $At(n)$ to which it belongs.

Definition 3.5. Given a logic program $\mathbb{P}$, a natural number n and an atom $A \in At(n)$, the *n-coinductive tree* for A in $\mathbb{P}$ is the possibly infinite tree T satisfying properties 1-4 of Definition 2.6 and property 5 replaced by the following[1]:

(5) For every $\wedge$-node s in T, let $A' \in At(n)$ be its label. For every clause $H \leftarrow B_1, \ldots, B_k$ of $\mathbb{P}$ and every term-matcher $\langle id_n, \tau \rangle$ of A' and H, with $B_1\tau, \ldots, B_k\tau \in At(n)$, s has exactly one child t, and viceversa. For each atom B in $\{B_1, \ldots, B_k\}\tau$, t has exactly one child labeled with B, and viceversa.

We recall from [32] the categorical formalization of this class of trees. The first step is to generalize the definition of the coalgebra p associated with a program $\mathbb{P}$. Definition 3.5 suggests how p should act on an atom $A \in At(n)$, for a fixed n:

$$A \mapsto \{\{B_1, \ldots, B_k\}\tau \mid H \leftarrow B_1, \ldots, B_k \text{ is a clause of } \mathbb{P},$$
$$A = H\tau \text{ and } B_1\tau, \ldots, B_k\tau \in At(n)\}. \tag{3.1}$$

For each clause $H \leftarrow B_1, \ldots, B_k$, there might be infinitely (but countably) many substitutions τ such that $A = H\tau$ (see e.g. [32]). Thus the object on the right-hand side of (3.1) will be associated with the functor $\mathcal{P}_c\mathcal{P}_f \colon \mathbf{Set} \to \mathbf{Set}$, where $\mathcal{P}_c$ and $\mathcal{P}_f$ are respectively the countable powerset functor and the finite powerset functor. In order to formalize this as a coalgebra on $At \colon \mathbf{L}_{\boldsymbol{\Sigma}}^{op} \to \mathbf{Set}$, let $\breve{\mathcal{P}}_c \colon \mathbf{Set}^{\mathbf{L}_{\boldsymbol{\Sigma}}^{op}} \to \mathbf{Set}^{\mathbf{L}_{\boldsymbol{\Sigma}}^{op}}$ and $\breve{\mathcal{P}}_f \colon \mathbf{Set}^{\mathbf{L}_{\boldsymbol{\Sigma}}^{op}} \to \mathbf{Set}^{\mathbf{L}_{\boldsymbol{\Sigma}}^{op}}$ be extensions of $\mathcal{P}_c$ and $\mathcal{P}_f$ respectively, given according to Definition 2.1. Then one would like to fix (3.1) as the definition of the n-component of a natural transformation $p \colon At \to \breve{\mathcal{P}}_c\breve{\mathcal{P}}_f(At)$. The key problem with this formulation is that p would *not* be a natural transformation, as shown by the following example.

Example 3.6. Let `NatList` be the same program of Example 3.4. Fix a substitution $\theta = \langle nil \rangle \colon 1 \to 0$ and, for each $n \in \mathbf{L}_{\boldsymbol{\Sigma}}^{op}$, suppose that $p(n) \colon At(n) \to \breve{\mathcal{P}}_c\breve{\mathcal{P}}_f(At)(n)$ is defined according to (3.1). Then the following square does not commute.

$$\begin{array}{ccc} At(1) & \xrightarrow{p(1)} & \breve{\mathcal{P}}_c\breve{\mathcal{P}}_f(At)(1) \\ {\scriptstyle At(\theta)} \downarrow & & \downarrow {\scriptstyle \breve{\mathcal{P}}_c\breve{\mathcal{P}}_f(At)(\theta)} \\ At(0) & \xrightarrow[p(0)]{} & \breve{\mathcal{P}}_c\breve{\mathcal{P}}_f(At)(0) \end{array}$$

A counterexample is provided by the atom `List`$(x_1) \in At(1)$. Passing through the bottom-left corner of the square, `List`(x_1) is mapped first to `List`$(nil) \in At(0)$ and then to $\{\emptyset\} \in \breve{\mathcal{P}}_c\breve{\mathcal{P}}_f(At)(0)$ - intuitively, this yields a refutation of the goal $\{$`List`$(x_1)\}$ with substitution of x_1 with nil. Passing through the top-right corner, `List`(x_1) is mapped first to $\emptyset \in \breve{\mathcal{P}}_c\breve{\mathcal{P}}_f(At)(1)$ and then to $\emptyset \in \breve{\mathcal{P}}_c\breve{\mathcal{P}}_f(At)(0)$, i.e. the computation ends up in a failure.

In [32, Sec.4] the authors overcome this difficulty by relaxing the naturality requirement. The morphism p is defined as a $\widetilde{\mathcal{P}_c}\widetilde{\mathcal{P}_f}$-coalgebra in the category $Lax(\mathbf{L}_{\boldsymbol{\Sigma}}^{op}, \mathbf{Poset})$ of locally ordered functors $\mathcal{F} \colon \mathbf{L}_{\boldsymbol{\Sigma}}^{op} \to \mathbf{Poset}$ and *lax* natural transformations, with each component $p(n)$ given according to (3.1) and $\widetilde{\mathcal{P}_c}\widetilde{\mathcal{P}_f}$ the extension of $\breve{\mathcal{P}}_c\breve{\mathcal{P}}_f$ to an endofunctor on $Lax(\mathbf{L}_{\boldsymbol{\Sigma}}^{op}, \mathbf{Poset})$.

---

[1] Our notion of coinductive tree corresponds to the notion of coinductive forest of breadth n as in [31, Def. 4.4], the only difference being that we "glue" together all trees of the forest into a single tree. It agrees with the definition given in [33, Def. 4.1] apart for the fact that the parameter n is fixed.

The lax approach fixes the problem, but presents also some drawbacks. Unlike the categories $\mathbf{Set}$ and $\mathbf{Set}^{\mathbf{L}_\Sigma^{op}}$, $Lax(\mathbf{L}_\Sigma^{op}, \mathbf{Poset})$ is neither complete nor cocomplete, meaning that a cofree comonad on $\widetilde{\mathcal{P}_c}\widetilde{\mathcal{P}_f}$ cannot be retrieved through the standard Constructions 3.1 and 3.2 that were used in the ground case. Moreover, the category of $\widetilde{\mathcal{P}_c}\widetilde{\mathcal{P}_f}$-coalgebrae becomes problematic, because coalgebra maps are subject to a commutativity property stricter than the one of lax natural transformations. These two issues force the formalization of non-ground logic program to use quite different (and more sophisticated) categorical tools than the ones employed for the ground case. Finally, as stressed in the Introduction, the laxness of p makes the resulting semantics not compositional.

4. Saturated Semantics

Motivated by the observations of the previous section, we propose a *saturated approach* to the semantics of logic programs. For this purpose, we consider an adjunction between presheaf categories as depicted on the left.

$$
\mathbf{Set}^{\mathbf{L}_\Sigma^{op}} \quad \underset{\mathcal{K}}{\overset{\mathcal{U}}{\underset{\longleftarrow}{\longrightarrow}}} \quad \bot \quad \mathbf{Set}^{|\mathbf{L}_\Sigma^{op}|}
\qquad\qquad
\begin{array}{ccc}
|\mathbf{L}_\Sigma^{op}| & \overset{\iota}{\hookrightarrow} & \mathbf{L}_\Sigma^{op} \\
{\scriptstyle \mathcal{F}}\downarrow & \swarrow {\scriptstyle \mathcal{K}(\mathcal{F})} & \\
\mathbf{Set} & &
\end{array}
$$

The left adjoint $\mathcal{U}$ is the forgetful functor, given by precomposition with the inclusion functor $\iota\colon |\mathbf{L}_\Sigma^{op}| \hookrightarrow \mathbf{L}_\Sigma^{op}$. As shown in [36, Th.X.1], $\mathcal{U}$ has a right adjoint $\mathcal{K}\colon \mathbf{Set}^{|\mathbf{L}_\Sigma^{op}|} \to \mathbf{Set}^{\mathbf{L}_\Sigma^{op}}$ sending $\mathcal{F}\colon |\mathbf{L}_\Sigma^{op}| \to \mathbf{Set}$ to its *right Kan extension* along ι. This is a presheaf $\mathcal{K}(\mathcal{F})\colon \mathbf{L}_\Sigma^{op} \to \mathbf{Set}$ mapping an object n of $\mathbf{L}_\Sigma^{op}$ to

$$\mathcal{K}(\mathcal{F})(n) \quad := \prod_{\theta \in \mathbf{L}_\Sigma^{op}[n,m]} \mathcal{F}(m)$$

where m is any object of $\mathbf{L}_\Sigma^{op}$. Intuitively, $\mathcal{K}(\mathcal{F})(n)$ is a set of tuples indexed by arrows with source n and such that, at index $\theta\colon n \to m$, there are elements of $\mathcal{F}(m)$. We use $\dot{x}\ \dot{y}, \ldots$ to denote such tuples and we write $\dot{x}(\theta)$ to denote the element at index θ of the tuple $\dot{x}$. Alternatively, when it is important to show how the elements depend from the indexes, we use $\langle x \rangle_{\theta:n\to m}$ (or simply $\langle x \rangle_\theta$) to denote the tuple having at index θ the element x. With this notation, we can express the behavior of $\mathcal{K}(\mathcal{F})\colon \mathbf{L}_\Sigma^{op} \to \mathbf{Set}$ on an arrow $\theta\colon n \to m$ as

$$\mathcal{K}(\mathcal{F})(\theta)\colon \dot{x} \mapsto \langle \dot{x}(\sigma \circ \theta) \rangle_{\sigma:m\to m'}. \tag{4.1}$$

The tuple $\langle \dot{x}(\sigma \circ \theta) \rangle_\sigma \in \mathcal{K}(\mathcal{F})(m)$ can be intuitively read as follows: for each $\sigma \in \mathbf{L}_\Sigma^{op}[m,m']$, the element indexed by σ is the one indexed by $\sigma \circ \theta \in \mathbf{L}_\Sigma^{op}[n,m']$ in the input tuple $\dot{x}$.

All this concerns the behavior of $\mathcal{K}$ on the objects of $\mathbf{Set}^{|\mathbf{L}_\Sigma^{op}|}$. For an arrow $f\colon \mathcal{F} \to \mathcal{G}$ in $\mathbf{Set}^{|\mathbf{L}_\Sigma^{op}|}$, the natural transformation $\mathcal{K}(f)$ is defined as an indexwise application of f on tuples from $\mathcal{K}(\mathcal{F})$. For all $n \in \mathbf{L}_\Sigma^{op}$, $\dot{x} \in \mathcal{K}(\mathcal{F})(n)$,

$$\mathcal{K}(f)(n)\colon \dot{x} \mapsto \langle f(m)(\dot{x}(\theta)) \rangle_{\theta:n\to m}.$$

For any presheaf $\mathcal{F}\colon \mathbf{L}_\Sigma^{op} \to \mathbf{Set}$, the unit η of the adjunction is instantiated to a morphism $\eta_\mathcal{F}\colon \mathcal{F} \to \mathcal{K}\mathcal{U}(\mathcal{F})$ given as follows: for all $n \in \mathbf{L}_\Sigma^{op}$, $X \in \mathcal{F}(n)$,

$$\eta_\mathcal{F}(n)\colon X \mapsto \langle \mathcal{F}(\theta)(X) \rangle_{\theta:n\to m}. \tag{4.2}$$

When taking $\mathcal{F}$ to be At, $\eta_{At}\colon At \to \mathcal{K}\mathcal{U}(At)$ maps an atom to its *saturation*: for each $A \in At(n)$, the tuple $\eta_{At}(n)(A)$ consists of all substitution instances $At(\theta)(A) = A\theta$ of A, each indexed by the corresponding $\theta \in \mathbf{L}_{\boldsymbol{\Sigma}}^{op}[n,m]$.

As shown in Example 3.6, given a program $\mathbb{P}$, the family of functions p defined by (3.1) fails to be a morphism in $\mathbf{Set}^{\mathbf{L}_{\boldsymbol{\Sigma}}^{op}}$. However, it forms a morphism in $\mathbf{Set}^{|\mathbf{L}_{\boldsymbol{\Sigma}}^{op}|}$

$$p\colon \mathcal{U}At \to \widehat{\mathcal{P}_c}\widehat{\mathcal{P}_f}(\mathcal{U}At)$$

where $\widehat{\mathcal{P}_c}$ and $\widehat{\mathcal{P}_f}$ denote the extensions of $\mathcal{P}_c$ and $\mathcal{P}_f$ to $\mathbf{Set}^{|\mathbf{L}_{\boldsymbol{\Sigma}}^{op}|}$, given as in Definition 2.1. The naturality requirement is trivially satisfied in $\mathbf{Set}^{|\mathbf{L}_{\boldsymbol{\Sigma}}^{op}|}$, since $|\mathbf{L}_{\boldsymbol{\Sigma}}^{op}|$ is discrete. The adjunction induces a morphism $p^{\sharp}\colon At \to \mathcal{K}\widehat{\mathcal{P}_c}\widehat{\mathcal{P}_f}\mathcal{U}(At)$ in $\mathbf{Set}^{\mathbf{L}_{\boldsymbol{\Sigma}}^{op}}$, defined as

$$At \xrightarrow{\;\eta_{At}\;} \mathcal{K}\mathcal{U}(At) \xrightarrow{\;\mathcal{K}(p)\;} \mathcal{K}\widehat{\mathcal{P}_c}\widehat{\mathcal{P}_f}\mathcal{U}(At). \tag{4.3}$$

In the sequel, we write $\mathcal{S}$ for $\mathcal{K}\widehat{\mathcal{P}_c}\widehat{\mathcal{P}_f}\mathcal{U}$. The idea is to let $\mathcal{S}$ play the same role as $\mathcal{P}_f\mathcal{P}_f$ in the ground case, with the coalgebra $p^{\sharp}\colon At \to \mathcal{S}(At)$ encoding the program $\mathbb{P}$. An atom $A \in At(n)$ is mapped to $\langle p(m)(A\sigma)\rangle_{\sigma\colon n\to m}$, that is:

$$p^{\sharp}(n)\colon A \mapsto \langle\{\{B_1,\ldots,B_k\}\tau \mid H \leftarrow B_1,\ldots,B_k \text{ is a clause of } \mathbb{P},$$
$$A\sigma = H\tau \text{ and } B_1\tau,\ldots,B_k\tau \in At(m)\}\rangle_{\sigma\colon n\to m}. \tag{4.4}$$

Intuitively, $p^{\sharp}(n)$ retrieves all unifiers $\langle\sigma,\tau\rangle$ of A and heads of $\mathbb{P}$: first, $A\sigma \in At(m)$ arises as a component of the saturation of A, according to $\eta_{At}(n)$; then, the substitution τ is given by term-matching on $A\sigma$, according to $K(p)(m)$.

By naturality of $p^{\sharp}$, we achieve the property of "commuting with substitutions" that was precluded by the term-matching approach, as shown by the following rephrasing of Example 3.6.

Example 4.1. Consider the same square of Example 3.6, with $p^{\sharp}$ in place of p and $\mathcal{S}$ in place of $\breve{\mathcal{P}_c}\breve{\mathcal{P}_f}$. The atom $\mathtt{List}(x_1) \in At(1)$ together with the substitution $\theta = \langle nil\rangle\colon 1 \to 0$ does not constitute a counterexample to commutativity anymore. Indeed $p^{\sharp}(1)$ maps $\mathtt{List}(x_1)$ to the tuple $\langle p(n)(\mathtt{List}(x_1)\sigma)\rangle_{\sigma\colon 1\to n}$, which is then mapped by $\mathcal{S}(At)(\theta)$ to $\langle p(n)(\mathtt{List}(x_1)\sigma' \circ \theta)\rangle_{\sigma'\colon 0\to n}$ according to (4.1). Observe that the latter is just the tuple $\langle p(n)(\mathtt{List}(nil)\sigma')\rangle_{\sigma'\colon 0\to n}$ obtained by applying first $At(\theta)$ and then $p^{\sharp}(0)$ to $\mathtt{List}(x_1)$.

Another benefit of saturated semantics is that $p^{\sharp}\colon At \to \mathcal{S}(At)$ lives in a (co)complete category which behaves (pointwise) as $\mathbf{Set}$. This allows us to follow the same steps as in the ground case, constructing a coalgebra for the cofree comonad $\mathcal{C}(\mathcal{S})$ as a straightforward generalization of Constructions 3.1 and 3.2. For this purpose, we first need to verify the following technical lemma.

Proposition 4.2. *The functor $\mathcal{S}$ is accessible and the terminal sequence for $At \times \mathcal{S}(\cdot)$ converges to a final $At \times \mathcal{S}(\cdot)$-coalgebra.*

Proof. By [52, Th.7], in order to show convergence of the terminal sequence it suffices to prove that $\mathcal{S}$ is an accessible mono-preserving functor. Since these properties are preserved by composition, we show them separately for each component of $\mathcal{S}$:

- Being adjoint functors between accessible categories, $\mathcal{K}$ and $\mathcal{U}$ are accessible themselves [3, Prop.2.23]. Moreover, they are both right adjoints: in particular, $\mathcal{U}$ is right adjoint to the left Kan extension functor along $\iota\colon |\mathbf{C}| \hookrightarrow \mathbf{C}$. It follows that both preserve limits, whence they preserve monos.

- Concerning functors $\widehat{\mathcal{P}_c}\colon \mathbf{Set}^{|\mathbf{L}_{\Sigma}^{op}|} \to \mathbf{Set}^{|\mathbf{L}_{\Sigma}^{op}|}$ and $\widehat{\mathcal{P}_f}\colon \mathbf{Set}^{|\mathbf{L}_{\Sigma}^{op}|} \to \mathbf{Set}^{|\mathbf{L}_{\Sigma}^{op}|}$, it is well-known that $\mathcal{P}_c\colon \mathbf{Set} \to \mathbf{Set}$ and $\mathcal{P}_f\colon \mathbf{Set} \to \mathbf{Set}$ are both mono-preserving accessible functors on $\mathbf{Set}$. It follows that $\widehat{\mathcal{P}_c}$ and $\widehat{\mathcal{P}_f}$ also have these properties, because (co)limits in presheaf categories are computed objectwise and monos are exactly the objectwise injective morphisms (as shown for instance in [37, Ch.6]). $\square$

Construction 4.3. The terminal sequence for the functor $At \times \mathcal{S}(\cdot)\colon \mathbf{Set}^{\mathbf{L}_{\Sigma}^{op}} \to \mathbf{Set}^{\mathbf{L}_{\Sigma}^{op}}$ consists of a sequence of objects X_α and arrows $\delta_\alpha\colon X_{\alpha+1} \to X_\alpha$, which are defined just as in Construction 3.1, with $\mathcal{S}$ replacing $\mathcal{P}_f\mathcal{P}_f$. By Proposition 4.2, this sequence converges to a limit X_γ such that $X_\gamma \cong X_{\gamma+1}$ and X_γ is the carrier of the cofree $\mathcal{S}$-coalgebra on At.

Since $\mathcal{S}$ is accessible (Proposition 4.2), the cofree comonad $\mathcal{C}(\mathcal{S})$ exists and maps At to X_γ given as in Construction 4.3. Below we provide a $\mathcal{C}(\mathcal{S})$-coalgebra structure $\llbracket \cdot \rrbracket_{p^\sharp}\colon At \to \mathcal{C}(\mathcal{S})(At)$ to At.

Construction 4.4. The terminal sequence for $At \times \mathcal{S}(\cdot)$ induces a cone $\{p_\alpha^\sharp\colon At \to X_\alpha\}_{\alpha<\gamma}$ as in Construction 3.2 with $p^\sharp$ and $\mathcal{S}$ replacing p and $\mathcal{P}_f\mathcal{P}_f$. This yields a natural transformation $\llbracket \cdot \rrbracket_{p^\sharp}\colon At \to X_\gamma$, where $X_\gamma = \mathcal{C}(\mathcal{S})(At)$.

As in the ground case, the coalgebra $\llbracket \cdot \rrbracket_{p^\sharp}$ is constructed as an iterative application of $p^\sharp$: we call *saturated $\wedge\vee$-tree* the associated tree structure.

Definition 4.5. Given a logic program $\mathbb{P}$, a natural number n and an atom $A \in At(n)$, the *saturated $\wedge\vee$-tree* for A in $\mathbb{P}$ is the possibly infinite tree T satisfying properties 1-3 of Definition 2.6 and properties 4 and 5 replaced by the following:

(4) Each $\vee$-node is labeled with a substitution σ and its children are $\wedge$-nodes.
(5) For every $\wedge$-node s in T, let $A' \in At(n')$ be its label. For every clause $H \leftarrow B_1, \ldots, B_k$ of $\mathbb{P}$ and every unifier $\langle\sigma, \tau\rangle$ of A' and H, with $\sigma\colon n' \to m'$ and $B_1\tau, \ldots, B_k\tau \in At(m')$, s has exactly one child t labeled with σ, and viceversa. For each atom B in $\{B_1, \ldots, B_k\}\tau$, t has exactly one child labeled with B, and viceversa.

We have now seen three kinds of tree, exhibiting different substitution mechanisms. In saturated $\wedge\vee$-trees one considers all the unifiers, whereas in $\wedge\vee$-trees and coinductive trees one restricts to most general unifiers and term-matchers respectively. Moreover, in a coinductive tree each $\wedge$-node is labeled with an atom in $At(n)$ for a fixed n, while in a saturated $\wedge\vee$-tree n can dynamically change.

Example 4.6. Part of the infinite saturated $\wedge\vee$-tree of $\texttt{List}(x_1) \in At(1)$ in $\texttt{NatList}$ is depicted below. Note that not all labels of $\wedge$-nodes belong to $At(1)$, as it would be the case for a coinductive tree: such information is inherited from the label of the parent $\vee$-node, which is now a substitution. For instance, both $\texttt{Nat}(x_1)$ and $\texttt{List}(x_2)$ belong to $At(2)$, since their parent is labeled with $\langle cons(x_1, x_2)\rangle\colon 1 \to 2$ (using the convention that the target of a substitution is the largest index appearing among its variables).
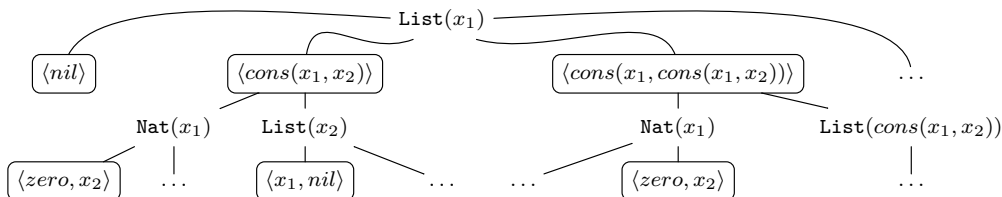
Next we verify that the notion of saturated $\wedge\vee$-tree is indeed the one given by saturated semantics. In the following proposition and in the rest of the paper, with an abuse of notation we use $\llbracket A \rrbracket_{p^\sharp}$ to denote the application of $\llbracket \cdot \rrbracket_{p^\sharp}(n)$ to $A \in At(n)$ without mentioning the object $n \in \mathbf{L}_{\mathbf{\Sigma}}^{op}$.

Proposition 4.7 (Adequacy). *For all n and $A \in At(n)$, the saturated $\wedge\vee$-tree of A in a program $\mathbb{P}$ is $\llbracket A \rrbracket_{p^\sharp}$.*

Proof. Fix $n \in \mathbf{L}_{\mathbf{\Sigma}}^{op}$. Just as Construction 3.1 allows to describe the cofree $\mathcal{P}_f \mathcal{P}_f$-coalgebra $\mathcal{C}(\mathcal{P}_f \mathcal{P}_f)(At)$ on At as the space of trees with two sorts of nodes occurring at alternating depth and one sort labelled by At, observing Construction 4.3 we can provide a similar description for the elements of $\mathcal{C}(\mathcal{S})(At)(n)$. Those are also trees with two sorts of nodes occurring at alternating depth. One sort (the one of $\wedge$-nodes), is labeled with elements of $At(m)$ for some $m \in \mathbf{L}_{\mathbf{\Sigma}}^{op}$. The root itself is an $\wedge$-node labeled with some atom in $At(n)$. The $\vee$-nodes children of an $\wedge$-node $B \in At(m)$ are not given by a plain set, as in the ground case, but instead by a tuple $\dot{x} \in \mathcal{S}(At)(m)$. We can represent $\dot{x}$ by drawing a child $\vee$-node t labeled with the substitution $\theta \colon m \to m'$ for each element S_t of the set $\dot{x}(\theta) \in \widehat{\mathcal{P}_c \mathcal{P}_f} At(m)$. The $\wedge$-node children of t are labeled with the elements of $S_t \in \widehat{\mathcal{P}_f} At(m')$.

The saturated $\wedge\vee$-tree for an atom $A \in At(n)$, as in Definition 4.5, is a tree of the above kind and thus an element of $\mathcal{C}(\mathcal{S})(At)(n)$. The function $A \mapsto T_A$ mapping A in its saturated $\wedge\vee$-tree T_A extends to an $At \times \mathcal{S}(\cdot)$-coalgebra morphism from At — with coalgebraic structure given by $\langle id, p^\sharp \rangle$ — to $\mathcal{C}(\mathcal{S})(At)$. By construction $\llbracket \cdot \rrbracket_{p^\sharp}$ is also an $At \times \mathcal{S}(\cdot)$-coalgebra morphism. Since $\mathcal{C}(\mathcal{S})(At)$ is the final $At \times \mathcal{S}(\cdot)$-coalgebra, these two morphisms must coincide, meaning that $\llbracket \cdot \rrbracket_{p^\sharp}$ maps A into its saturated $\wedge\vee$-tree T_A. $\square$

For an arrow $\theta \in \mathbf{L}_{\mathbf{\Sigma}}^{op}[n, m]$, we write $\overline{\theta}$ for $\mathcal{C}(\mathcal{S})(At)(\theta) \colon \mathcal{C}(\mathcal{S})(At)(n) \to \mathcal{C}(\mathcal{S})(At)(m)$. With this notation, we can state the compositionality result motivating our approach. Its proof is an immediate consequence of the naturality of $\llbracket \cdot \rrbracket_{p^\sharp}$.

Theorem 4.8 (Compositionality). *For all atoms $A \in At(n)$ and substitutions $\theta \in \mathbf{L}_{\mathbf{\Sigma}}^{op}[n, m]$,*

$$\llbracket A\theta \rrbracket_{p^\sharp} = \llbracket A \rrbracket_{p^\sharp} \overline{\theta}.$$

We conclude this section with a concrete description of the behavior of the operator $\overline{\theta}$, for a given substitution $\theta \in \mathbf{L}_{\mathbf{\Sigma}}^{op}[n, m]$. Let r be the root of a tree $T \in \mathcal{C}(\mathcal{S})(At)(n)$ and r' the root of $T\overline{\theta}$. Then

(1) the node r has label A iff r' has label $A\theta$;
(2) the node r has a child t with label $\sigma \circ \theta$ and children $t_1, \ldots, t_n$ iff r' has a child t' with label σ and children $t_1 \ldots t_n$.

Note that the children $t_1, \ldots, t_n$ are exactly the same in both trees: $\overline{\theta}$ only modifies the root and the $\vee$-nodes at depth 1 of T, while it leaves untouched all the others. This peculiar behavior can be better understood by observing that the definition of $\mathcal{K}(\mathcal{F})(\theta)$, as in (4.1), is independent of the presheaf $\mathcal{F}$. As a result, $\overline{\theta} = X_\gamma(\theta)$ is independent of all the X_αs built in Construction 4.3.

Example 4.9. Recall from Example 4.6 the saturated $\wedge\vee$-tree $[\![\texttt{List}(x_1)]\!]_{p^\sharp}$. For $\theta = \langle cons(x_1, x_2) \rangle$, the tree $[\![\texttt{List}(x_1)]\!]_{p^\sharp} \overline{\theta}$ is depicted below.
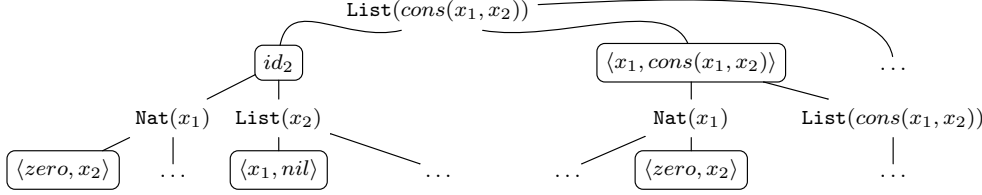
5. DESATURATION

One of the main features of coinductive trees is to represent (sound) and-or parallel derivations of goals. This leads the authors of [31] to a resolution algorithm exploiting the two forms of parallelism. Motivated by these developments, we include coinductive trees in our framework, showing how they can be obtained as a "desaturation" of saturated $\wedge\vee$-trees.

For this purpose, the key ingredient is given by the *counit* ϵ of the adjunction $\mathcal{U} \dashv \mathcal{K}$. Given a presheaf $\mathcal{F} \colon |\mathbf{L}_{\boldsymbol{\Sigma}}^{op}| \to \mathbf{Set}$, the morphism $\epsilon_\mathcal{F} \colon \mathcal{U}\mathcal{K}(\mathcal{F}) \to \mathcal{F}$ is defined as follows: for all $n \in \mathbf{L}_{\boldsymbol{\Sigma}}^{op}$ and $\dot{x} \in \mathcal{U}\mathcal{K}(\mathcal{F})(n)$,

$$\epsilon_\mathcal{F}(n) \colon \dot{x} \mapsto \dot{x}(id_n) \tag{5.1}$$

where $\dot{x}(id_n)$ is the element of the input tuple $\dot{x}$ which is indexed by the identity substitution $id_n \in \mathbf{L}_{\boldsymbol{\Sigma}}^{op}[n, n]$. In the logic programming perspective, the intuition is that, while the unit of the adjunction provides the saturation of an atom, the counit reverses the process. It takes the saturation of an atom and gives back the substitution instance given by the identity, that is, the atom itself.

We now want to define a "desaturation" map $\overline{d}$ from saturated $\wedge\vee$-trees to coinductive trees, acting as a pointwise application of $\epsilon_{\mathcal{U}At}$. For this purpose, first we state the construction of the cofree comonad on $\widehat{\mathcal{P}_c\mathcal{P}_f}$ for later reference.

Construction 5.1. The terminal sequence for $\mathcal{U}At \times \widehat{\mathcal{P}_c\mathcal{P}_f}(\cdot) \colon \mathbf{Set}^{|\mathbf{L}_{\boldsymbol{\Sigma}}^{op}|} \to \mathbf{Set}^{|\mathbf{L}_{\boldsymbol{\Sigma}}^{op}|}$ consists of sequences of objects Y_α and arrows $\xi_\alpha \colon Y_{\alpha+1} \to Y_\alpha$, defined by induction on α as follows.

$$Y_\alpha := \begin{cases} \mathcal{U}At & \alpha = 0 \\ \mathcal{U}At \times \widehat{\mathcal{P}_c\mathcal{P}_f}(Y_\beta) & \alpha = \beta + 1 \end{cases} \qquad \xi_\alpha := \begin{cases} \pi_1 & \alpha = 0 \\ id_{\mathcal{U}At} \times \widehat{\mathcal{P}_c\mathcal{P}_f}(\xi_\beta) & \alpha = \beta + 1 \end{cases}$$

For α a limit ordinal, Y_α and ξ_α are defined as expected. As stated in the proof of Proposition 4.2, $\widehat{\mathcal{P}_c\mathcal{P}_f}$ is a mono-preserving accessible functors. Then by [52, Th.7] we know that the sequence given above converges to a limit Y_χ such that $Y_\chi \cong Y_{\chi+1}$ and Y_χ is the value of $\mathcal{C}(\widehat{\mathcal{P}_c\mathcal{P}_f}) \colon \mathbf{Set}^{|\mathbf{L}_{\boldsymbol{\Sigma}}^{op}|} \to \mathbf{Set}^{|\mathbf{L}_{\boldsymbol{\Sigma}}^{op}|}$ on $\mathcal{U}At$, where $\mathcal{C}(\widehat{\mathcal{P}_c\mathcal{P}_f})$ is the cofree comonad on $\widehat{\mathcal{P}_c\mathcal{P}_f}$ induced by the terminal sequence given above, analogously to Construction 3.1.

The next construction defines the desidered morphism $\overline{d} \colon \mathcal{U}(\mathcal{C}(\mathcal{S})(At)) \to \mathcal{C}(\widehat{\mathcal{P}_c\mathcal{P}_f})(\mathcal{U}At)$.

Construction 5.2. Consider the image of the terminal sequence converging to $\mathcal{C}(\mathcal{S})(At) = X_\gamma$ (Construction 4.3) under the forgetful functor $\mathcal{U} \colon \mathbf{Set}^{\mathbf{L}_{\boldsymbol{\Sigma}}^{op}} \to \mathbf{Set}^{|\mathbf{L}_{\boldsymbol{\Sigma}}^{op}|}$. We define a

sequence of natural transformations $\{d_\alpha \colon \mathcal{U}(X_\alpha) \to Y_\alpha\}_{\alpha < \gamma}$ as follows[2]:

$$d_\alpha \quad := \quad \begin{cases} id_{\mathcal{U}At} & \alpha = 0 \\ id_{\mathcal{U}At} \times \left(\widehat{\mathcal{P}_c\mathcal{P}_f}(d_\beta) \circ \epsilon_{\widehat{\mathcal{P}_c\mathcal{P}_f}\mathcal{U}(X_\beta)} \right) & \alpha = \beta + 1. \end{cases}$$

$$
\begin{array}{ccc}
\mathcal{U}(X_\beta) & \xrightarrow{\quad d_\beta \quad} & Y_\beta \\
\uparrow{\scriptstyle \mathcal{U}(\delta_\beta)} & & \uparrow{\scriptstyle \xi_\beta} \\
\mathcal{U}(X_{\beta+1}) & \xrightarrow{\quad d_{\beta+1} \quad} & Y_{\beta+1}
\end{array}
$$

$$id_{\mathcal{U}At} \times \epsilon_{\widehat{\mathcal{P}_c\mathcal{P}_f}\mathcal{U}(X_\beta)} \qquad id_{\mathcal{U}At} \times \widehat{\mathcal{P}_c\mathcal{P}_f}(d_\beta)$$

$$\mathcal{U}At \times \widehat{\mathcal{P}_c\mathcal{P}_f}\mathcal{U}(X_\beta)$$

For $\alpha < \gamma$ a limit ordinal, an arrow $d_\alpha \colon \mathcal{U}(X_\alpha) \to Y_\alpha$ is provided by the limiting property of Y_α. In order to show that the limit case is well defined, observe that, for every $\beta < \alpha$, the above square commutes, that is, $\xi_\beta \circ d_{\beta+1} = d_\beta \circ \mathcal{U}(\delta_\beta)$. This can be easily checked by ordinal induction, using the fact that $\epsilon_{\widehat{\mathcal{P}_c\mathcal{P}_f}\mathcal{U}(X_\beta)}$ is a natural transformation for each $\beta < \alpha$.

We now turn to the definition of a natural transformation $\overline{d} \colon \mathcal{U}\big(\mathcal{C}(\mathcal{S})(At)\big) \to \mathcal{C}(\widehat{\mathcal{P}_c\mathcal{P}_f})(\mathcal{U}At)$. If $\chi \leq \gamma$, then this is provided by $d_\chi \colon \mathcal{U}(X_\chi) \to Y_\chi$ together with the limiting property of $\mathcal{U}(X_\gamma)$ on $\mathcal{U}(X_\chi)$. In case $\gamma < \chi$, observe that, since X_γ is isomorphic to $X_{\gamma+1}$, then X_γ is isomorphic to X_ζ for all $\zeta > \gamma$, and in particular $X_\gamma \cong X_\chi$. Then we can suitably extend the sequence to have a natural transformation $d_\chi \colon \mathcal{U}(X_\chi) \to Y_\chi$. The morphism $\overline{d}$ is given as the composition of d_χ with the isomorphism between $\mathcal{U}(X_\gamma)$ and $\mathcal{U}(X_\chi)$.

The next theorem states that $\overline{d}$ is a translation from saturated to coinductive trees: given an atom $A \in At(n)$, it maps $[\![A]\!]_{p\sharp}$ to the n-coinductive tree of A. The key intuition is that n-coinductive trees can be seen as saturated $\wedge\vee$-trees where the labeling of $\vee$-nodes has been restricted to the identity substitution id_n, represented as $\bullet$ (see Definition 3.5). The operation of pruning all $\vee$-nodes (and their descendants) in $[\![A]\!]_{p\sharp}$ which are not labeled with id_n is precisely what is provided by Construction 5.2, in virtue of the definition of the counit ϵ given in (5.1).

Theorem 5.3 (Desaturation). *Let $[\![\cdot]\!]_{p\sharp} \colon At \to \mathcal{C}(\mathcal{S})(At)$ be defined for a logic program $\mathbb{P}$ according to Construction 4.4 and $\overline{d} \colon \mathcal{U}\big(\mathcal{C}(\mathcal{S})(At)\big) \to \mathcal{C}(\widehat{\mathcal{P}_c\mathcal{P}_f})(\mathcal{U}At)$ be defined according to Construction 5.2. Then for all $n \in |\mathbf{L}_\Sigma^{op}|$ and $A \in \mathcal{U}At(n)$, the n-coinductive tree of A in $\mathbb{P}$ is $\big(\overline{d} \circ \mathcal{U}([\![\cdot]\!]_{p\sharp})\big)(n)(A)$.*

Theorem 5.3 provides an alternative formalization for the coinductive tree semantics [32], given by composition of the saturated semantics with desaturation. In fact it represents a different approach to the non-compositionality problem: instead of relaxing naturality to lax naturality, we simply forget about all the arrows of the index category $\mathbf{L}_\Sigma^{op}$, shifting the framework from $\mathbf{Set}^{\mathbf{L}_\Sigma^{op}}$ to $\mathbf{Set}^{|\mathbf{L}_\Sigma^{op}|}$. The substitutions on trees (that are essential, for

---

[2]Concerning the successor case, observe that $id_{\mathcal{U}At} \times \big(\widehat{\mathcal{P}_c\mathcal{P}_f}(d_\beta) \circ \epsilon_{\widehat{\mathcal{P}_c\mathcal{P}_f}\mathcal{U}(X_\beta)} \big)$ is in fact an arrow from $\mathcal{U}At \times \mathcal{U}\mathcal{K}\widehat{\mathcal{P}_c\mathcal{P}_f}\mathcal{U}(X_\beta)$ to $Y_{\beta+1}$. However, the former is isomorphic to $\mathcal{U}(X_{\beta+1}) = \mathcal{U}\big(At \times \mathcal{K}\widehat{\mathcal{P}_c\mathcal{P}_f}\mathcal{U}(X_\beta)\big)$, because $\mathcal{U}$ is a right adjoint (as observed in Proposition 4.2) and thence it commutes with products.

instance, for the resolution algorithm given in [31]) exist at the saturated level, i.e. in $\mathcal{C}(\mathcal{S})(At)$, and they are given precisely as the operator $\overline{\theta}$ described at the end of Section 4.

Example 5.4. The coinductive tree for $\mathtt{List}(cons(x_1, x_2))$ in $\mathtt{NatList}$ is depicted on the right. It is constructed by desaturating the tree $[\![\mathtt{List}(cons(x_1, x_2))]\!]_{p^\sharp}$ in Example 4.9, i.e., by pruning all the $\vee$-nodes (and their descendants) that are not labeled with id_2.
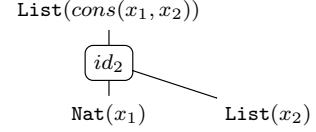
6. SOUNDNESS AND COMPLETENESS

The notion of coinductive tree leads to a semantics that is sound and complete with respect to SLD-resolution [31, Th.4.8]. To this aim, a key role is played by *derivation subtrees* of coinductive trees: they are defined exactly as derivation subtrees of $\wedge\vee$-trees (Definition 2.7) and represent SLD-derivations where the computation only advances by term-matching.

Similarly, we now want to define a notion of subtree for saturated semantics. This requires care: saturated $\wedge\vee$-trees are associated with unification, which is more liberal than term-matching. In particular, like $\wedge\vee$-trees, they may represent unsound derivation strategies (*cf.* Example 3.4). However, in saturated $\wedge\vee$-trees *all* unifiers, not just the most general ones, are taken into account. This gives enough flexibility to shape a sound notion of subtree, based on an implicit synchronization of the substitutions used in different branches.

Definition 6.1. Let T be the saturated $\wedge\vee$-tree for an atom A in a program $\mathbb{P}$. A subtree T' of T is called a *synched derivation subtree* if it satisfies properties 1-3 of Definition 2.7 and the following condition:

(4) all $\vee$-nodes of T' at the same depth are labeled with the same substitution.

A *synched refutation subtree* is a finite synched derivation subtree with only $\vee$-nodes as leaves. Its *answer* is the substitution $\theta_{2k+1} \circ \ldots \theta_3 \circ \theta_1$, where θ_i is the (unique) substitution labeling the $\vee$-nodes of depth i and $2k + 1$ is its maximal depth.

The prefix "synched" emphasizes the restriction to and-parallelism encoded in Definition 6.1. Intuitively, we force all subgoals at the same depth to proceed with *the same* substitution. For instance, this rules out the unsound derivation of Example 3.4. For a comparison with Definition 2.7, note that derivation subtrees can be seen as special instances of synched derivation subtrees where all the substitutions are forced to be identities.

We are now in position to compare the different semantics for logic programming.

Theorem 6.2 (Soundness and Completeness)**.** *Let $\mathbb{P}$ be a logic program and $A \in At(n)$ an atom. The following are equivalent.*

(1) *The saturated $\wedge\vee$-tree for A in $\mathbb{P}$ has a synched refutation subtree with answer θ.*
(2) *For some $m \in \mathbf{L}_\Sigma^{op}$, the m-coinductive tree for $A\theta$ in $\mathbb{P}$ has a refutation subtree.*
(3) *There is an SLD-refutation for $\{A\}$ in $\mathbb{P}$ with correct answer θ.*

In statement (3), note that θ is the *correct* (and not the computed) answer: indeed, the unifiers associated with each derivation step in the synched refutation subtree are not necessarily the most general ones. Towards a proof of Theorem 6.2, a key role is played by the following proposition.
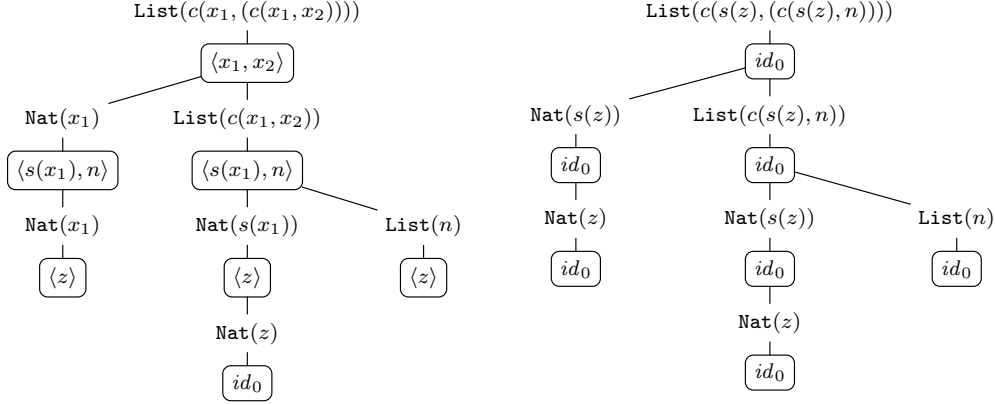
Figure 1: Successful synched derivation subtrees for $\texttt{List}(cons(x_1, (cons(x_1, x_2))))$ (left) and $\texttt{List}(cons(succ(zero), (cons(succ(zero), nil))))$ (right) in $\texttt{NatList}$. The symbols $cons$, nil, $succ$ and $zero$ are abbreviated to c, n, s and z respectively.

Proposition 6.3. *Let $\mathbb{P}$ be a logic program and $A \in At(n)$ an atom. If $\llbracket A \rrbracket_{p^\sharp}$ has a synched refutation subtree with answer $\theta \colon n \to m$, then $\llbracket A \rrbracket_{p^\sharp} \bar{\theta}$ has a synched refutation subtree whose $\vee$-nodes are all labeled with id_m.*

Proof. See Appendix A. □

Figure 1 provides an example of the construction needed for Proposition 6.3. Note that the root of the rightmost tree is labeled with an atom of the form $A\theta$, where θ and A are respectively the answer and the label of the root of the leftmost tree. The rightmost tree is a refutation subtree of the 0-coinductive tree for $A\theta$ and can be obtained from the leftmost tree via a procedure involving the operator $\bar{\theta}$ discussed at the end of Section 4.

We are now ready to provide a proof of Theorem 6.2 combining Proposition 6.3, the desaturation procedure of Theorem 5.3 and the compositionality result of Theorem 4.8.

Proof of Theorem 6.2. The statement $(2 \Leftrightarrow 3)$ is a rephrasing of [31, Th.4.8], whence we focus on proving $(1 \Leftrightarrow 2)$, where m is always the target object of θ.

$(1 \Rightarrow 2)$. If $\llbracket A \rrbracket_{p^\sharp}$ has a synched refutation subtree with answer θ, then by Proposition 6.3, $\llbracket A \rrbracket_{p^\sharp} \bar{\theta}$ has a synched refutation subtree T whose $\vee$-nodes are all labeled with id_m. Compositionality (Theorem 4.8) guarantees that T is a synched refutation subtree of $\llbracket A\theta \rrbracket_{p^\sharp}$. Since all the $\vee$-nodes of T are labeled with id_m, T is preserved by desaturation. This means that T is a refutation subtree of $\bar{d}(\mathcal{U}(\llbracket \cdot \rrbracket_{p^\sharp}))(m)(A\theta)$ which, by Theorem 5.3, is the m-coinductive tree for $A\theta$ in $\mathbb{P}$.

$(2 \Leftarrow 1)$. If the m coinductive tree for $A\theta$ has a refutation subtree T then, by Theorem 5.3, this is also the case for $\bar{d}(\mathcal{U}(\llbracket \cdot \rrbracket_{p^\sharp}))(m)(A\theta)$. This means that T is a synched derivation subtree of $\llbracket A\theta \rrbracket_{p^\sharp}$ whose $\vee$-nodes are all labeled by id_m. By compositionality, T is also a subtree of $\llbracket A \rrbracket_{p^\sharp} \bar{\theta}$. Let t be the $\vee$-node at the first depth of T. By construction of the operator $\bar{\theta}$, the root of $\llbracket A \rrbracket_{p^\sharp}$ has a child t' labeled with θ having the same children as t. Therefore $\llbracket A \rrbracket_{p^\sharp}$ has a synched refutation subtree with answer θ. □

7. Bialgebraic Semantics of Goals: the Ground Case

In this section we lay the foundations of a (saturated) semantics that applies directly to goals. As outlined in the introduction, a main motivation for this extension is the study of yet another form of compositionality, now with respect to the algebraic structure of goals.

First, we approach the question in the simpler case of ground logic programs. Such a program $\mathbb{P}$, that in Section 3.1 has been represented as a $\mathcal{P}_f\mathcal{P}_f$-coalgebra $p\colon At \to \mathcal{P}_f\mathcal{P}_f(At)$, will now be modeled as a certain bialgebra. The coalgebraic part will be given by the endofunctor $\mathcal{P}_f$: this corresponds to the outer occurrence of $\mathcal{P}_f$ in $p\colon At \to \mathcal{P}_f\mathcal{P}_f(At)$ and encodes non-determinism, i.e., the possibility that an atom matches multiple heads in a program. The algebraic part, encoding the internal structure of a goal, instead will be given by a monad $\mathcal{T}$ that corresponds to the inner occurrence of $\mathcal{P}_f$ in $p\colon At \to \mathcal{P}_f\mathcal{P}_f(At)$. Intuitively, $\mathcal{T}(At)$ represents a goal, i.e. a collection of atoms, to be processed *in parallel*.

To formally define the type of our bialgebrae, we need to provide a distributive law $\lambda\colon \mathcal{T}\mathcal{P}_f \Rightarrow \mathcal{P}_f\mathcal{T}$. Our specification for λ stems from the observation that $\mathcal{P}_f(At)$ models a *disjunction* of atoms, whereas $\mathcal{T}(At)$ models a *conjunction*. Then λ should just distribute conjunction over disjunction, for instance:

$$(A_1 \vee A_2) \wedge (B_1 \vee B_2) \mapsto (A_1 \wedge B_1) \vee (A_1 \wedge B_2) \vee (A_2 \wedge B_1) \vee (A_2 \wedge B_2). \qquad (7.1)$$

Provided this specification for λ, one could naively think of modelling $\mathcal{T}$ as $\mathcal{P}_f$ itself, so that the conjunction $\mathcal{T}(At)$ is represented as a finite set of atoms. However, this would pose a problem with the naturality of λ. Consider again the example above, now formalized with $\mathcal{T} = \mathcal{P}_f$, and suppose to check naturality for $f\colon At \to At$ defined as $(A_1 \mapsto B_1, A_2 \mapsto B_2)$:

$$
\begin{array}{ccc}
\mathcal{T}\mathcal{P}_f(At) & \xrightarrow{\lambda_{At}} & \mathcal{P}_f\mathcal{T}(At) \\
{\scriptstyle \mathcal{T}\mathcal{P}_f(f)} \downarrow & & \downarrow {\scriptstyle \mathcal{P}_f\mathcal{T}(f)} \\
\mathcal{T}\mathcal{P}_f(At) & \xrightarrow{\lambda_{At}} & \mathcal{P}_f\mathcal{T}(At)
\end{array}
\qquad (7.2)
$$

The function $\lambda_{At} \circ \mathcal{T}\mathcal{P}_f(f)$ maps $\{\{A_1, A_2\}, \{B_1, B_2\}\}$ first to $\{\{B_1, B_2\}\}$ and then to $\{\{B_1\}, \{B_2\}\}$. Instead $\mathcal{P}_f\mathcal{T}(f) \circ \lambda_{At}$ maps $\{\{A_1, A_2\}, \{B_1, B_2\}\}$ first to $\{\{A_1, B_1\}, \{A_1, B_2\}, \{A_2, B_1\}, \{A_2, B_2\}\}$ and then to $\{\{B_1\}, \{B_1, B_2\}, \{B_2\}\}$. Therefore our specification (7.1) for λ, with $\mathcal{T}$ modeled as $\mathcal{P}_f$, would not yield a natural transformation.

For this reason, we model instead the elements of $\mathcal{T}(At)$ as *lists* of atoms. Formally, we let $\mathcal{T}$ be the *list* endofunctor $\mathcal{L}\colon \mathbf{Set} \to \mathbf{Set}$ mapping a set X into the set of lists of elements of X and a function $f\colon X \to Y$ into its componentwise application to lists in $\mathcal{L}(X)$. Such a functor forms a monad with unit $\eta_X^{\mathcal{L}}\colon X \Rightarrow \mathcal{L}X$ given by $x \mapsto [x]$ (where $[x]$ is the list with a single element x) and a multiplication given by flattening: for $l_1, \ldots, l_n$ elements of $\mathcal{L}(X)$, $\mu_X^{\mathcal{L}}\colon \mathcal{L}\mathcal{L}X \Rightarrow \mathcal{L}X$ maps $[l_1, \ldots, l_n]$ to $l_1 :: \ldots :: l_n$, where $::$ denotes list concatenation. In virtue of its definition, we will make use of the notation $::$ for the function $\mu_X^{\mathcal{L}}$, whenever X is clear from the context.

There is a distributive law $\lambda\colon \mathcal{L}\mathcal{P}_f \Rightarrow \mathcal{P}_f\mathcal{L}$ of the monad $\mathcal{P}_f$ over the monad $\mathcal{L}$ given by assignments $[X_1, \ldots, X_n] \mapsto \{[x_1, \ldots, x_n] \mid x_i \in X_i\}$ (*cf.* [38, Ex. 2.4.8]). One can readily check that this definition implements the specification given in (7.1):

$$[\{A_1, A_2\}, \{B_1, B_2\}] \mapsto \{[A_1, B_1], [A_1, B_2], [A_2, B_1], [A_2, B_2]\}.$$

Also, the counterexample to commutativity of (7.2) given above is neutralized as follows.

$$
\begin{array}{ccc}
[\{A_1, A_2\}, \{B_1, B_2\}] & \xrightarrow{\;\;\lambda_{At}\;\;} & \{[A_1, B_1], [A_1, B_2], [A_2, B_1], [A_2, B_2]\} \\
{\scriptstyle \mathcal{T}\mathcal{P}_f(f)} \downarrow & & \downarrow {\scriptstyle \mathcal{P}_f\mathcal{T}(f)} \\
[\{B_1, B_2\}, \{B_1, B_2\}] & \xrightarrow{\;\;\lambda_{At}\;\;} & \{[B_1, B_1], [B_1, B_2], [B_2, B_1], [B_2, B_2]\}
\end{array}
$$

Remark 7.1. Non-commutativity of diagram (7.2) is essentially due to the idempotence of the powerset constructor. Thus, in order to achieve naturality, multisets instead of lists would also work, as both structures are not idempotent. We chose lists in conformity with implementations of SLD-resolution (for instance PROLOG) where atoms are processed following their order in the goal.

Convention 7.2. As motivated above, we will develop our framework modeling goals as lists instead of sets. This requires a mild reformulation of the constructions presented in Section 3.1, with $\mathcal{L}$ taking the role of the inner occurrence of $\mathcal{P}_f$: throughout this and the next section we suppose that a ground logic program $\mathbb{P}$ is encoded as a coalgebra $p\colon At \to \mathcal{P}_f\mathcal{L}(At)$ (instead of $p\colon At \to \mathcal{P}_f\mathcal{P}_f(At)$) and its semantics as a map $[\![\cdot]\!]_p\colon At \to \mathcal{C}(\mathcal{P}_f\mathcal{L})(At)$. It is immediate to see that this is an harmless variation on the framework for coalgebraic logic programming developed so far and all our results still hold. Accordingly, $\wedge\vee$-trees are now elements of $\mathcal{C}(\mathcal{P}_f\mathcal{L})(At)$ (instead of $\mathcal{C}(\mathcal{P}_f\mathcal{P}_f)(At)$ as in Section 3) and therefore the children of an $\vee$-node form a list rather than a set. For that reason, while keeping the standard representation of $\wedge\vee$-trees, we shall implicitly assume a left to right ordering amongst children of $\vee$-nodes (*cf.* Example 8.7).

We now have all the formal ingredients to define the extension from coalgebraic to bialgebraic operational semantics.

Construction 7.3. Let $\mathbb{P}$ be a ground logic program and $p\colon At \to \mathcal{P}_f\mathcal{L}(At)$ be the coalgebra associated with $\mathbb{P}$. We define the $\mathcal{P}_f$-coalgebra $p_\lambda\colon \mathcal{L}At \to \mathcal{P}_f\mathcal{L}At$ in the way prescribed by Proposition 2.3 as

$$
\mathcal{L}At \xrightarrow{\;\mathcal{L}p\;} \mathcal{L}\mathcal{P}_f\mathcal{L}At \xrightarrow{\;\lambda_{\mathcal{L}At}\;} \mathcal{P}_f\mathcal{L}\mathcal{L}At \xrightarrow{\;\mathcal{P}_f(::)\;} \mathcal{P}_f\mathcal{L}At \; .
$$

By Proposition 2.3, $(\mathcal{L}At, ::, p_\lambda)$ forms a λ-bialgebra.

Remark 7.4. Equivalently, the map $p_\lambda\colon \mathcal{L}At \to \mathcal{P}_f\mathcal{L}At$ may be obtained with the following universal construction. Let $\mathcal{U}^{\mathcal{L}} \dashv \mathcal{F}^{\mathcal{L}}$ be the canonical adjunction between **Set** and the category $\mathbf{EM}(\mathcal{L})$ of Eilenberg-Moore algebrae for the monad $\mathcal{L}$. By using the distributive law λ we can define an $\mathcal{L}$-algebra structure on $\mathcal{P}_f\mathcal{L}At$ as $h := \mathcal{P}_f(::) \circ \lambda_{\mathcal{L}At}\colon \mathcal{L}\mathcal{P}_f\mathcal{L}At \to \mathcal{P}_f\mathcal{L}At$. Let $\widetilde{p}\colon (\mathcal{L}At, ::) \to (\mathcal{P}_f\mathcal{L}At, h)$ be the unique extension of $p\colon At \to \mathcal{P}_f\mathcal{L}At$ to an algebra morphism in $\mathbf{EM}(\mathcal{L})$ along the adjunction $\mathcal{U}^{\mathcal{L}} \dashv \mathcal{F}^{\mathcal{L}}$. The map $p_\lambda\colon \mathcal{L}At \to \mathcal{P}_f\mathcal{L}At$ of Construction 7.3 is simply $\mathcal{U}^{\mathcal{L}}(\widetilde{p})$. By definition it makes the following diagram commute.

$$
\begin{array}{ccc}
At & \xrightarrow{\;\eta^{\mathcal{L}}_{At}\;} & \mathcal{L}At \\
{\scriptstyle p} \downarrow & \swarrow {\scriptstyle p_\lambda} & \\
\mathcal{P}_f\mathcal{L}At & &
\end{array}
\tag{7.3}
$$

As investigated in [47], the same pattern that here leads to p_λ from a given p has many relevant instances. Most notably, the standard powerset construction for non-deterministic automata can be presented in a bialgebraic setting, as a mapping $t \mapsto \bar{t}$, where t encodes a non-deterministic automaton on state space X, $\bar{t}$ a deterministic one on state space $\mathcal{P}X$ and a diagram analogous to (7.3) yields $\bar{t} \circ \eta_X^{\mathcal{P}} = t$. It is worth noticing that the powerset construction performs a *determinization* of the automaton t: non-determinism is internalized, that is, an element of the state space $\mathcal{P}X$ models a *disjunction* of states. Our construction follows a different intuition: the non-determinism given by clauses with the same head is preserved and what is internalized is the possible $\wedge$-*parallel behavior* of a computation: indeed, an element of $\mathcal{L}At$ models a *conjunction* of atoms.

Example 7.5. Let $p_\lambda \colon \mathcal{L}At \to \mathcal{P}_f\mathcal{L}At$ be constructed out of the coalgebra $p \colon At \to \mathcal{P}_f\mathcal{L}At$ associated with the logic program of Example 3.3. For a concrete grasp on Construction 7.3, we compute $p_\lambda([\mathsf{p}(b,b), \mathsf{p}(b,c)]) \in \mathcal{P}_f\mathcal{L}At$. First, observe that $p \colon At \to \mathcal{P}_f\mathcal{L}At$ assigns to the atom $\mathsf{p}(b,b)$ the set $\{[\mathsf{q}(c)], [\mathsf{p}(b,a), \mathsf{p}(b,c)]\}$ and to $\mathsf{p}(b,c)$ the set $\{[\mathsf{q}(a), \mathsf{q}(b), \mathsf{q}(c)]\}$. It is therefore easy to see that $\mathcal{L}p$ maps $[\mathsf{p}(b,b), \mathsf{p}(b,c)]$ into the list of sets of lists

$$[\{[\mathsf{q}(c)], [\mathsf{p}(b,a), \mathsf{p}(b,c)]\}, \ \{[\mathsf{q}(a), \mathsf{q}(b), \mathsf{q}(c)]\}].$$

This is mapped by $\lambda_{\mathcal{L}At}$ into the set of lists of lists

$$\{[[\mathsf{q}(c)], [\mathsf{q}(a), \mathsf{q}(b), \mathsf{q}(c)]], \ [[\mathsf{p}(b,a), \mathsf{p}(b,c)], [\mathsf{q}(a), \mathsf{q}(b), \mathsf{q}(c)]]\}$$

and finally, via $\mathcal{P}_f(::)$ into the set of lists

$$\{[\mathsf{q}(c), \mathsf{q}(a), \mathsf{q}(b), \mathsf{q}(c)], \ [\mathsf{p}(b,a), \mathsf{p}(b,c), \mathsf{q}(a), \mathsf{q}(b), \mathsf{q}(c)]\}$$

that is the value $p_\lambda([\mathsf{p}(b,b), \mathsf{p}(b,c)])$.

The operational reading of such a computation is that $[\mathsf{p}(b,b), \mathsf{p}(b,c)]$ is a goal given by the conjunction of atoms $\mathsf{p}(b,b)$ and $\mathsf{p}(b,c)$. The action of p_λ tells us that, in order to refute $[\mathsf{p}(b,b), \mathsf{p}(b,c)]$, one has to refute either $[\mathsf{q}(c), \mathsf{q}(a), \mathsf{q}(b), \mathsf{q}(c)]$ (the first element of $p_\lambda([\mathsf{p}(b,b), \mathsf{p}(b,c)])$) *or* $[\mathsf{p}(b,a), \mathsf{p}(b,c), \mathsf{q}(a), \mathsf{q}(b), \mathsf{q}(c)]$ (the second element of $p_\lambda([\mathsf{p}(b,b), \mathsf{p}(b,c)])$).

For later use, it is interesting to observe also that $p_\lambda([\,]) = \{[\,]\}$ for any program p: this is because $[\,] \overset{\mathcal{L}p}{\mapsto} [\,] \overset{\lambda_{\mathcal{L}At}}{\mapsto} \{[\,]\} \overset{\mathcal{P}_f(::)}{\mapsto} \{[\,]\}$.

Remark 7.6. For yet another view on Construction 7.3, we remark that the map p_λ can be presented in terms of the following SOS-rules, where $l \to l'$ means $l' \in p_\lambda(l)$.

$$l1 \ \frac{l \in p(A)}{[A] \to l} \qquad l2 \ \frac{l_1 \to l_1' \quad l_2 \to l_2'}{l_1 :: l_2 \to l_1' :: l_2'} \qquad l3 \ \frac{}{[\,] \to [\,]}$$

Rule ($l1$) corresponds to commutativity of (7.3). Rule ($l2$) states that p_λ preserves the algebraic structure of $\mathcal{L}At$ given by list concatenation $::$. Operationally, this means that one step of (parallel) resolution for $l_1 :: l_2$ requires both a step for l_1 and a step for l_2. Finally, rule ($l3$) encodes the trivial behavior of p_λ on the empty list, as observed in Example 7.5.

We now provide a cofree construction for the semantics $[\![\cdot]\!]_{p_\lambda}$ arising from p_λ.

Construction 7.7. Let $\mathcal{C}(\mathcal{P}_f)(\mathcal{L}At)$ denote the cofree $\mathcal{P}_f$-coalgebra on $\mathcal{L}At$, obtained analogously to Construction 3.1. It enjoys a final $\mathcal{L}At \times \mathcal{P}_f(\cdot)$-coalgebra structure $c \colon \mathcal{C}(\mathcal{P}_f)(\mathcal{L}At) \overset{\cong}{\to} \mathcal{L}At \times \mathcal{P}_f(\mathcal{C}(\mathcal{P}_f)(\mathcal{L}At))$. We now want to lift its universal property to the setting of bialgebrae, showing that $\mathcal{C}(\mathcal{P}_f)(\mathcal{L}At)$ forms a final λ'-bialgebra, where $\lambda' \colon \mathcal{L}(\mathcal{L}At \times \mathcal{P}_f(\cdot)) \Rightarrow$

$\mathcal{L}At \times \mathcal{P}_f\mathcal{L}(\cdot)$ is a distributive law of the monad $\mathcal{L}$ over the functor $\mathcal{L}At \times \mathcal{P}_f(\cdot)$ defined by

$$\mathcal{L}(\mathcal{L}At \times \mathcal{P}_f(X)) \xrightarrow{\langle \mathcal{L}\pi_1, \mathcal{L}\pi_2 \rangle} \mathcal{L}\mathcal{L}At \times \mathcal{L}\mathcal{P}_f(X) \xrightarrow{:: \times \lambda_X} \mathcal{L}At \times \mathcal{P}_f\mathcal{L}(X) \ .$$

For this purpose, we apply the construction of Proposition 2.4. First, using finality of $c \colon \mathcal{C}(\mathcal{P}_f)(\mathcal{L}At) \xRightarrow{\cong} \mathcal{L}At \times \mathcal{P}_f(\mathcal{C}(\mathcal{P}_f)(\mathcal{L}At))$ we define an $\mathcal{L}$-algebra $\overline{::} \colon \mathcal{L}(\mathcal{C}(\mathcal{P}_f)(\mathcal{L}At)) \to \mathcal{C}(\mathcal{P}_f)(\mathcal{L}At)$ as follows:

$$
\begin{array}{ccc}
\mathcal{L}(\mathcal{C}(\mathcal{P}_f)(\mathcal{L}At)) \dashrightarrow{\overline{::}} \longrightarrow & \mathcal{C}(\mathcal{P}_f)(\mathcal{L}At) \\
\ \downarrow{\scriptstyle \mathcal{L}c} & \ \downarrow{\scriptstyle c} \\
\mathcal{L}(\mathcal{L}At \times \mathcal{P}_f(\mathcal{C}(\mathcal{P}_f)(\mathcal{L}At))) & \\
\ \downarrow{\scriptstyle \lambda'_{\mathcal{C}(\mathcal{P}_f)(\mathcal{L}At)}} & \\
\mathcal{L}At \times \mathcal{P}_f(\mathcal{L}(\mathcal{C}(\mathcal{P}_f)(\mathcal{L}At))) \xrightarrow{id_{\mathcal{L}At} \times \mathcal{P}_f(\overline{::})} & \mathcal{L}At \times \mathcal{P}_f(\mathcal{C}(\mathcal{P}_f)(\mathcal{L}At))
\end{array}
$$

This algebraic structure yields the final λ'-bialgebra $\big(\mathcal{C}(\mathcal{P}_f)(\mathcal{L}At), \overline{::}, c\big)$ as guaranteed by Proposition 2.4. Now we turn to the definition of the semantics $\llbracket \cdot \rrbracket_{p_\lambda} \colon \mathcal{L}At \to \mathcal{C}(\mathcal{P}_f)(\mathcal{L}At)$. First, observe that the λ-bialgebra $(\mathcal{L}At, ::, p_\lambda)$ canonically extends to a λ'-bialgebra by letting $\langle id_{\mathcal{L}At}, p_\lambda \rangle$ be the $\mathcal{L}At \times \mathcal{P}_f(\cdot)$-coalgebra structure on $\mathcal{L}At$. We can then use finality of $\big(\mathcal{C}(\mathcal{P}_f)(\mathcal{L}At), \overline{::}, c\big)$ to obtain the unique λ'-bialgebra morphism $\llbracket \cdot \rrbracket_{p_\lambda} \colon \mathcal{L}At \to \mathcal{C}(\mathcal{P}_f)(\mathcal{L}At)$ making the following diagram commute:

$$
\begin{array}{ccc}
\mathcal{L}\mathcal{L}At \xrightarrow{\ \ \ \ \mathcal{L}\llbracket \cdot \rrbracket_{p_\lambda} \ \ \ \ } & \mathcal{L}(\mathcal{C}(\mathcal{P}_f)(\mathcal{L}At)) \\
\ \downarrow{\scriptstyle ::} & \ \downarrow{\scriptstyle \overline{::}} \\
\mathcal{L}At \dashrightarrow{\llbracket \cdot \rrbracket_{p_\lambda}} \dashrightarrow & \mathcal{C}(\mathcal{P}_f)(\mathcal{L}At) \\
\ \downarrow{\scriptstyle \langle id_{\mathcal{L}At}, p_\lambda \rangle} & \ \downarrow{\scriptstyle c} \\
\mathcal{L}At \times \mathcal{P}_f(\mathcal{L}At) \xrightarrow{id_{\mathcal{L}At} \times \mathcal{P}_f(\llbracket \cdot \rrbracket_{p_\lambda})} & \mathcal{L}At \times \mathcal{P}_f(\mathcal{C}(\mathcal{P}_f)(\mathcal{L}At)).
\end{array}
\tag{7.4}
$$

One can check that the above construction of the map $\llbracket \cdot \rrbracket_{p_\lambda}$ can be equivalently obtained by building a cone with vertex $\mathcal{L}At$ on the terminal sequence generating $\mathcal{C}(\mathcal{P}_f)(\mathcal{L}At)$, analogously to Construction 3.2.

Since $\llbracket \cdot \rrbracket_{p_\lambda}$ is given as a morphism of bialgebrae instead of plain coalgebrae, it will preserve the algebraic structure of $\mathcal{L}At$. This allow us to state the motivating compositionality property of bialgebraic semantics where, intuitively, list concatenation :: models the conjunction $\wedge$ described in the introduction.

Theorem 7.8 ($\wedge$-Compositionality). *Given lists $l_1, \ldots, l_k$ of atoms in At:*

$$\llbracket l_1 :: \ldots :: l_k \rrbracket_{p_\lambda} = \llbracket l_1 \rrbracket_{p_\lambda} \overline{::} \ldots \overline{::} \llbracket l_k \rrbracket_{p_\lambda}.$$

Where $l_1 :: \ldots :: l_k$ is notation for $::([l_1, \ldots, l_k]) = \mu^{\mathcal{L}}_{At}([l_1, \ldots, l_k])$ and $\llbracket l_1 \rrbracket_{p_\lambda} \overline{::} \ldots \overline{::} \llbracket l_k \rrbracket_{p_\lambda}$ for $\overline{::}([\llbracket l_1 \rrbracket_{p_\lambda}, \ldots, \llbracket l_k \rrbracket_{p_\lambda}]).$

Proof. The statement is given by the following derivation:

$$\llbracket l_1 :: \ldots :: l_k \rrbracket_{p_\lambda} = \llbracket \cdot \rrbracket_{p_\lambda} \circ :: ([l_1, \ldots, l_k])$$
$$= \bar{::} \circ \mathcal{L} \llbracket \cdot \rrbracket_{p_\lambda} ([l_1, \ldots, l_k])$$
$$= \bar{::} (\llbracket l_1 \rrbracket_{p_\lambda}, \ldots, \llbracket l_k \rrbracket_{p_\lambda}])$$
$$= \llbracket l_1 \rrbracket_{p_\lambda} \bar{::} \ldots \bar{::} \llbracket l_k \rrbracket_{p_\lambda}.$$

The first and the last equality are just given by unfolding notation. The second equality amounts to commutativity of the top square in diagram (7.4) and the third one is given by definition of $\mathcal{L}$ on the function $\llbracket \cdot \rrbracket_{p_\lambda}$. $\qquad \square$

Example 7.9. Recall from Example 7.5, that $p_\lambda([\mathsf{p}(b, b)]) = \{[\mathsf{q}(c)], [\mathsf{p}(b, a), \mathsf{p}(b, c)]\}$ and $p_\lambda([\mathsf{p}(b, c)]) = \{[\mathsf{q}(a), \mathsf{q}(b), \mathsf{q}(c)]\}$. Below we represent the values $\llbracket \mathsf{p}(b, b) \rrbracket_{p_\lambda}$ (on the left) and $\llbracket \mathsf{p}(b, c) \rrbracket_{p_\lambda}$ (on the right) as trees.
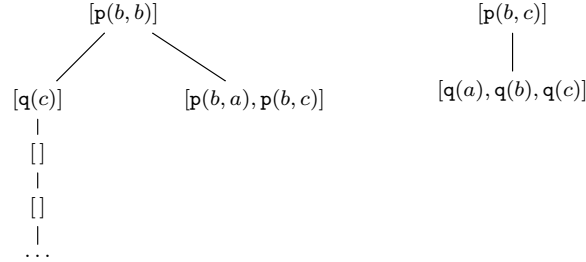

The idea is that edges represent the transitions of the rule system of Remark 7.6. The node $[\mathsf{q}(c)]$ has a child labeled with the empty list, since $p(\mathsf{q}(c)) = \{[\,]\}$ (*cf.* Example 3.3) and thus, by virtue of rule ($l1$), $p_\lambda([\mathsf{q}(c)]) = \{[\,]\}$. Instead, $[\mathsf{q}(a), \mathsf{q}(b), \mathsf{q}(c)]$ has no children because $p_\lambda([\mathsf{q}(a), \mathsf{q}(b), \mathsf{q}(c)])$ is empty. That follows by the fact that $p_\lambda(\mathsf{q}(a))$ — and, in fact, also $p_\lambda(\mathsf{q}(b))$ — is empty and thus we cannot trigger rule ($l2$) to let $[\mathsf{q}(a), \mathsf{q}(b), \mathsf{q}(c)]$ make a transition. Intuitively, this means that $[\mathsf{q}(a), \mathsf{q}(b), \mathsf{q}(c)]$ cannot be refuted because not all of its atoms have a refutation. A similar consideration holds for $[\mathsf{p}(b, a), \mathsf{p}(b, c)]$.

In Example 7.5, we have shown that

$$p_\lambda([\mathsf{p}(b, b), \mathsf{p}(b, c)]) = \{[\mathsf{q}(c), \mathsf{q}(a), \mathsf{q}(b), \mathsf{q}(c)], [\mathsf{p}(b, a), \mathsf{p}(b, c), \mathsf{q}(a), \mathsf{q}(b), \mathsf{q}(c)]\}$$

and, by similar arguments to those above, we can show that both $p_\lambda([\mathsf{q}(c), \mathsf{q}(a), \mathsf{q}(b), \mathsf{q}(c)])$ and $p_\lambda([\mathsf{p}(b, a), \mathsf{p}(b, c), \mathsf{q}(a), \mathsf{q}(b), \mathsf{q}(c)])$ are empty. Therefore, we can depict $\llbracket \mathsf{p}(b, b), \mathsf{p}(b, c) \rrbracket_{p_\lambda}$ as the tree below.
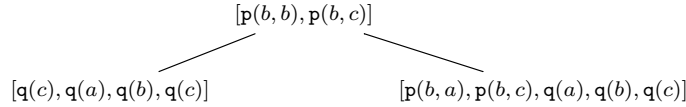
By virtue of Theorem 7.8, such a tree can be computed also by concatenating via $\bar{::}$ the tree $\llbracket [\mathsf{p}(b, b)] \rrbracket_{p_\lambda}$ with the tree $\llbracket [\mathsf{p}(b, c)] \rrbracket_{p_\lambda}$ depicted above. The operation $\bar{::}$ on trees T_1 and T_2 can also be described as follows.

(1) If the root of T_1 has label l_1 and the root of T_2 has label l_2, then the root of $T_1 \bar{::} T_2$ has label $l_1 :: l_2$;

(2) If T_1 has a child T_1' and T_2 has a child T_2', then $T_1 \bar{::} T_2$ has a child $T_1' \bar{::} T_2'$.

Observe that such trees are rather different from the $\wedge\vee$-trees introduced in Definition 2.6 (*cf.* also Example 3.3): all nodes are of the same kind (there is no more distinction between

$\wedge$-nodes and $\vee$-nodes) and are labeled by lists of atoms (rather than just atoms). In the next section, we will formally introduce such trees under the name of (parallel) $\vee$-trees and show that they provide a sound and complete semantics for ground logic programs.

8. Soundness and Completeness of Bialgebraic Ground Semantics

In this section we investigate the relation between the semantics $[\![\cdot]\!]_p$ and $[\![\cdot]\!]_{p_\lambda}$, stating a relative soundness and completeness result.

In Section 3.1 we observed that $[\![\cdot]\!]_p$ — of type $At \to \mathcal{C}(\mathcal{P}_f\mathcal{L})(At)$, following Convention 7.2 — maps an atom A into its $\wedge\vee$-tree (Definition 2.6). As a first step of our analysis, we provide an analogous operational understanding for the value $[\![l]\!]_{p_\lambda} \in \mathcal{C}(\mathcal{P}_f)(\mathcal{L}At)$ associated with a goal $l \in \mathcal{L}At$. The resulting notion of $\vee$-tree will correspond to the one intuitively given in Example 7.9.

Definition 8.1. Given a ground logic program $\mathbb{P}$ and a list $l \in \mathcal{L}At$ of atoms, the *(parallel)* $\vee$-*tree* for l in $\mathbb{P}$ is the possibly infinite tree T satisfying the following properties:
  (1) Each node in T is labeled with a list of atoms and the root is labeled with l.
  (2) Let s be a node in T with label $l' = [A_1, \dots, A_k]$. For every list $[C_1, \dots, C_k]$ of clauses of $\mathbb{P}$ such that $H_i = A_i$ for each $C_i = H^i \leftarrow B_1^i, \dots, B_j^i$, s has exactly one child t, and viceversa. The node t is labeled with the list $l_1 :: \dots :: l_k$, where $l_i = [B_1^i, \dots, B_j^i]$ is the body of clause C_i.

Differently from $\wedge\vee$-trees, where two kinds of nodes yield a distinction between or- and and-parallelism, $\vee$-trees have only one kind of nodes, intuitively corresponding to or-parallelism. The and-parallelism, which in $\wedge\vee$-trees is given by the branching of and-nodes labeled with an atom, is encoded in $\vee$-trees by the labeling of nodes with lists of atoms. The children of a node labeled with l yield the result of simultaneously matching *each* atom in l with heads in the program (*cf.* rule ($l2$) in Remark 7.6).[3]

Analogously to Proposition 4.7, it is immediate to check the following observation.

Proposition 8.2 (Adequacy). *Given a list of atoms $l \in \mathcal{L}At$ and a program $\mathbb{P}$, $[\![l]\!]_{p_\lambda} \in \mathcal{C}(\mathcal{P}_f)(\mathcal{L}At)$ is the $\vee$-tree for l in $\mathbb{P}$.*

We are now in position to provide a translation between the two notions of tree associated respectively with the semantics $[\![\cdot]\!]_p$ and $[\![\cdot]\!]_{p_\lambda}$.

Construction 8.3. There is a canonical representation of $\wedge\vee$-trees as $\vee$-trees given as follows. First recall that the domain $\mathcal{C}(\mathcal{P}_f\mathcal{L})(At)$ of $\wedge\vee$-trees is the final $At \times \mathcal{P}_f\mathcal{L}(\cdot)$-coalgebra, say with structure given by $u\colon \mathcal{C}(\mathcal{P}_f\mathcal{L})(At) \xrightarrow{\cong} At \times \mathcal{P}_f\mathcal{L}\big(\mathcal{C}(\mathcal{P}_f\mathcal{L})(At)\big)$ (*cf.* Construction 3.1). Let $d := (\eta_{At}^{\mathcal{L}} \times id) \circ u$ be the extension of u to an $\mathcal{L}At \times \mathcal{P}_f\mathcal{L}(\cdot)$-coalgebra. We now define an $\mathcal{L}At \times \mathcal{P}_f(\cdot)$-coalgebra structure $d_{\lambda'}$ on $\mathcal{L}(\mathcal{C}(\mathcal{P}_f\mathcal{L})(At))$ in the way prescribed

---

[3]This synchronous form of computation is what makes $\vee$-trees essentially different from SLD-trees [35], in which also nodes are labeled with lists, but the parent-child relation describes the unification of a *single* atom in the parent node.

by Proposition 2.3:

$$
\begin{array}{ccc}
\mathcal{L}(\mathcal{C}(\mathcal{P}_f\mathcal{L})(At)) & \xrightarrow{\ \ d_{\lambda'}\ \ } & \mathcal{L}At \times \mathcal{P}_f\mathcal{L}\big(\mathcal{C}(\mathcal{P}_f\mathcal{L})(At)\big) \\
{\scriptstyle \mathcal{L}d}\big\downarrow & {\scriptstyle \underset{\cdot\cdot}{\parallel}} & \big\uparrow{\scriptstyle id_{\mathcal{L}At}\times\mathcal{P}_f(::)} \\
\mathcal{L}\big(\mathcal{L}At \times \mathcal{P}_f\mathcal{L}\big(\mathcal{C}(\mathcal{P}_f\mathcal{L})(At)\big)\big) & \xrightarrow{\ \lambda'_{\mathcal{C}(\mathcal{P}_f\mathcal{L})(At)}\ } & \mathcal{L}At \times \mathcal{P}_f\mathcal{L}\mathcal{L}\big(\mathcal{C}(\mathcal{P}_f\mathcal{L})(At)\big)
\end{array}
$$

Then we use finality of $\mathcal{C}(\mathcal{P}_f)(\mathcal{L}At)$ (the domain of $\vee$-trees) to obtain our representation map $r := \llbracket\cdot\rrbracket_{d_{\lambda'}} \circ \eta^{\mathcal{L}}_{\mathcal{C}(\mathcal{P}_f\mathcal{L})(At)}$ as in the following commutative diagram.

$$(8.1)$$

The representation map r is a well-behaved translation between the semantics given by $\llbracket\cdot\rrbracket_p$ and the one given by $\llbracket\cdot\rrbracket_{p_\lambda}$, as shown by the following property.

Proposition 8.4. *For all $A \in At$, $r(\llbracket A\rrbracket_p) = \llbracket[A]\rrbracket_{p_\lambda}$.*

Proof. By construction $\llbracket\cdot\rrbracket_p\colon At \to \mathcal{C}(\mathcal{P}_f\mathcal{L})(At)$ is a $\mathcal{P}_f\mathcal{L}$-coalgebra morphism and thus clearly also an $\mathcal{L}At\times\mathcal{P}_f\mathcal{L}(\cdot)$-coalgebra morphism. By Proposition 2.3 the canonical mapping of an $\mathcal{L}At \times \mathcal{P}_f\mathcal{L}(\cdot)$-coalgebra to a λ'-bialgebra

$$
\begin{array}{ccc}
At & & \mathcal{L}\mathcal{L}At \\
{\scriptstyle \langle\eta^{\mathcal{L}}_{At},p\rangle}\big\downarrow & \mapsto & \big\downarrow{\scriptstyle ::} \\
& & \mathcal{L}At \\
& & \big\downarrow{\scriptstyle \langle\eta^{\mathcal{L}}_{At},p\rangle_{\lambda'}} \\
\mathcal{L}At \times \mathcal{P}_f\mathcal{L}(At) & & \mathcal{L}At \times \mathcal{P}_f\mathcal{L}At
\end{array}
$$

$$
\begin{array}{ccc}
\mathcal{C}(\mathcal{P}_f\mathcal{L})(At) & & \mathcal{L}\mathcal{L}\mathcal{C}(\mathcal{P}_f\mathcal{L})(At) \\
{\scriptstyle d}\big\downarrow & \mapsto & \big\downarrow{\scriptstyle ::} \\
& & \mathcal{L}\mathcal{C}(\mathcal{P}_f\mathcal{L})(At) \\
& & \big\downarrow{\scriptstyle d_{\lambda'}} \\
\mathcal{L}At \times \mathcal{P}_f\mathcal{L}(\mathcal{C}(\mathcal{P}_f\mathcal{L})(At)) & & \mathcal{L}At \times \mathcal{P}_f\mathcal{L}\mathcal{C}(\mathcal{P}_f\mathcal{L})(At)
\end{array}
$$

is functorial, meaning that $\mathcal{L}\llbracket\cdot\rrbracket_p\colon \mathcal{L}At \to \mathcal{L}(\mathcal{C}(\mathcal{P}_f\mathcal{L})(At))$ is a λ'-bialgebra morphism and thus in particular a morphism of $\mathcal{L}At \times \mathcal{P}_f(\cdot)$-coalgebrae. By construction $\llbracket\cdot\rrbracket_{p_\lambda}$ and $\llbracket\cdot\rrbracket_{d_{\lambda'}}$ are also $\mathcal{L}At \times \mathcal{P}_f(\cdot)$-coalgebra morphisms. Therefore $\llbracket\cdot\rrbracket_{d_{\lambda'}} \circ \mathcal{L}\llbracket\cdot\rrbracket_p = \llbracket\cdot\rrbracket_{p_\lambda}$ by finality of

$\mathcal{C}(\mathcal{P}_f)(\mathcal{L}At)$. Precomposing both sides with $\eta^{\mathcal{L}}_{At}$ yields the statement of the proposition:

$$\llbracket \cdot \rrbracket_{p_\lambda} \circ \eta^{\mathcal{L}}_{At} = \llbracket \cdot \rrbracket_{d_{\chi'}} \circ \mathcal{L}\llbracket \cdot \rrbracket_p \circ \eta^{\mathcal{L}}_{At} = \llbracket \cdot \rrbracket_{d_{\chi'}} \circ \eta^{\mathcal{L}}_{\mathcal{C}(\mathcal{P}_f\mathcal{L})(At)} \circ \llbracket \cdot \rrbracket_p = r \circ \llbracket \cdot \rrbracket_p.$$

$\square$

In order to study soundness and completeness of $\llbracket \cdot \rrbracket_{p_\lambda}$, we give a notion of refutation for ∨-trees.

Definition 8.5. Let $T \in \mathcal{C}(\mathcal{P}_f)(\mathcal{L}At)$ be an ∨-tree. A *derivation subtree* of T is a sequence of nodes $s_1, s_2, \ldots$ such that s_1 is the root of T and s_{i+1} is a child of s_i. A *refutation subtree* is a finite derivation subtree $s_1, \ldots, s_n$ where the last node s_n is labeled with the empty list.

In fact, derivation subtrees of ∨-trees have no branching: they are just paths starting from the root. This is coherent with the previously introduced notions of subtree (*cf.* Definition 2.7): there the only branching allowed was given by and-parallelism, which in the case of ∨-tress has been internalized inside the node labels. For refutation subtrees, the intuition is that they represents paths of computation where all atoms in the initial goal $[A_1, \ldots, A_n]$ have been refuted, whence eventually the current goal becomes the empty list.

Theorem 8.6 (Soundness and Completeness)**.** *Let $A_1, \ldots, A_n$ be atoms in At. Then $\llbracket [A_1, \ldots, A_n] \rrbracket_{p_\lambda}$ has a refutation subtree if and only if $\llbracket A_i \rrbracket_p$ has a refutation subtree for each A_i.*

Proof. The statement is given by the following derivation:

$\llbracket A_i \rrbracket_p$ has a refutation subtree for each A_i iff $r(\llbracket A_i \rrbracket_p)$ has a refutation subtree for each A_i

iff $\llbracket [A_i] \rrbracket_{p_\lambda}$ has a refutation subtree for each A_i

iff $\llbracket [A_1] \rrbracket_{p_\lambda} \,\overline{::}\, \ldots \,\overline{::}\, \llbracket [A_n] \rrbracket_{p_\lambda}$ has a refutation subtree

iff $\llbracket [A_1, \ldots, A_n] \rrbracket_{p_\lambda}$ has a refutation subtree

The first and the third equivalence are given by checking that the property of having a refutation subtree is preserved and reflected both by the representation map $r \colon \mathcal{C}(\mathcal{P}_f\mathcal{L})(At) \to \mathcal{C}(\mathcal{P}_f)(\mathcal{L}At)$ (*cf.* Construction 8.3) and by the concatenation operation $\overline{::}$ on ∨-trees. The second equivalence is given by Proposition 8.4 and the last one by Theorem 7.8. $\square$

Example 8.7. Consider the ground logic program in Example 3.3 and compare the ∧∨-tree $\llbracket p(b,b) \rrbracket_p$ with the ∨-tree $\llbracket [p(b,b)] \rrbracket_{p_\lambda}$ — that is, $r(\llbracket p(b,b) \rrbracket_p)$ — depicted below on the left and right, respectively.

Following Convention 7.2, children of ∨-nodes form a list which is implicit in the representation above: for instance, in the tree on the left $p(b,a)$ and $p(b,c)$ are displayed according to their order in $[p(b,a), p(b,c)] \in p(p(b,b))$.

This convention is instrumental in illustrating the behaviour of the representation map r: all the children $t_1, \ldots, t_k$ of an $\vee$-nodes (labeled with $\bullet$) of an $\wedge\vee$-trees T are grouped in $r(T)$ into a single node whose label lists all the atoms labeling $t_1, \ldots, t_k$. For instance, the rightmost child of $[\![\mathtt{p}(b,b)]\!]_p$ becomes in $[\![[\mathtt{p}(b,b)]]\!]_{p_\lambda}$ a node labeled with $[\mathtt{p}(b,a), \mathtt{p}(b,c)]$. It is also worth to note that the whole subtree reachable from $[\![\mathtt{p}(b,c)]\!]_p$ is pruned: since $p_\lambda(\mathtt{p}(b,a))$ is the empty set, then also $p_\lambda([\mathtt{p}(b,a), \mathtt{p}(b,c)])$ is empty. Intuitively, $\mathtt{p}(b,c)$ should be proved in conjunction with $\mathtt{p}(b,a)$ which has no proof and therefore the (parallel) resolution of $[\mathtt{p}(b,a), \mathtt{p}(b,c)]$ can stop immediately.

Instead, the unique refutation subtree of $[\![\mathtt{p}(b,b)]\!]_p$ (that is $\mathtt{p}(b,b) - \bullet - \mathtt{q}(c) - \bullet$) is preserved in $r([\![\mathtt{p}(b,b)]\!]_p)$. Indeed, $[\mathtt{p}(b,b)] - [\mathtt{q}(c)] - [\,]$ is a refutation subtree of $[\![\mathtt{p}(b,b)]\!]_{p_\lambda}$.

9. BIALGEBRAIC SEMANTICS OF GOALS: THE GENERAL CASE

In the sequel we generalize the bialgebraic semantics for goals from ground to arbitrary logic programs. Our approach is to extend the saturated semantics $[\![\cdot]\!]_{p^\sharp}$ in $\mathbf{Set}^{\mathbf{L}_\Sigma^{op}}$ introduced in Section 4. This will yield compositionality both with respect to the substitutions in the index category $\mathbf{L}_\Sigma^{op}$ (*cf.* Theorem 4.8) and with respect to the concatenation of goals (extending the result of Theorem 7.8).

Convention 9.1. As we did for ground programs (*cf.* Convention 7.2), also for arbitrary logic programs we intend to model goals as lists of atoms. This requires a mild reformulation of our framework for saturated semantics. For this purpose, we introduce the extension $\widehat{\mathcal{L}} \colon \mathbf{Set}^{|\mathbf{L}_\Sigma^{op}|} \to \mathbf{Set}^{|\mathbf{L}_\Sigma^{op}|}$ of $\mathcal{L} \colon \mathbf{Set} \to \mathbf{Set}$ as in Definition 2.1. Since $\mathcal{L}$ is a monad, then $\widehat{\mathcal{L}}$ is also a monad by Proposition 2.2. A logic program $\mathbb{P}$ is now encoded in $\mathbf{Set}^{|\mathbf{L}_\Sigma^{op}|}$ as a coalgebra $p \colon \mathcal{U}At \to \widehat{\mathcal{P}_c \mathcal{L}} \mathcal{U}(At)$ (instead of $p \colon \mathcal{U}At \to \widehat{\mathcal{P}_c \mathcal{P}_f} \mathcal{U}(At)$) and its saturation in $\mathbf{Set}^{\mathbf{L}_\Sigma^{op}}$ as a colgebra $p^\sharp \colon At \to \mathcal{K} \widehat{\mathcal{P}_c \mathcal{L}} \mathcal{U}(At)$. The saturated semantics of $\mathbb{P}$ (Construction 4.4) is given by $[\![\cdot]\!]_{p^\sharp} \colon At \to \mathcal{C}(\mathcal{K} \widehat{\mathcal{P}_c \mathcal{L}} \mathcal{U})(At)$. All results stated in Section 4, 5 and 6 continue to hold in this setting.

Our first task is to generalize Construction 7.3. The extension $p_\delta^\sharp$ of a saturated logic program $p^\sharp \colon At \to \mathcal{K} \widehat{\mathcal{P}_c \mathcal{L}} \mathcal{U}(At)$ to a bialgebra will depend on a distributive law δ, yet to be defined. The domain of $p_\delta^\sharp$ will be the presheaf $\check{\mathcal{L}}At \in \mathbf{Set}^{\mathbf{L}_\Sigma^{op}}$ of goals, where $\check{\mathcal{L}} \colon \mathbf{Set}^{\mathbf{L}_\Sigma^{op}} \to \mathbf{Set}^{\mathbf{L}_\Sigma^{op}}$ extends $\mathcal{L} \colon \mathbf{Set} \to \mathbf{Set}$ as in Definition 2.1. Since $\mathcal{L}$ is a monad, by Proposition 2.2 $\check{\mathcal{L}}$ is also a monad with unit and multiplication given componentwise by the ones of $\mathcal{L}$.

For ground logic programs, the definition of a bialgebra involves the construction of a $\mathcal{P}_f$-coalgebra out of a $\mathcal{P}_f \mathcal{L}$-coalgebra. For arbitrary logic programs there is a type mismatch, because in $p^\sharp \colon At \to \mathcal{K} \widehat{\mathcal{P}_c \mathcal{L}} \mathcal{U}(At)$ the functor $\check{\mathcal{L}}$ does not apply to At. However, this can be easily overcome by observing the following general property of the adjunction $\mathcal{U} \dashv \mathcal{K}$.

Proposition 9.2. *Let* $\mathcal{F} \colon \mathbf{Set} \to \mathbf{Set}$ *be a functor and* $\widehat{\mathcal{F}} \colon \mathbf{Set}^{|\mathbf{L}_\Sigma^{op}|} \to \mathbf{Set}^{|\mathbf{L}_\Sigma^{op}|}$, $\check{\mathcal{F}} \colon \mathbf{Set}^{\mathbf{L}_\Sigma^{op}} \to \mathbf{Set}^{\mathbf{L}_\Sigma^{op}}$ *its extensions to presheaf categories, given as in Definition 2.1. Then* $\widehat{\mathcal{F}} \mathcal{U} = \mathcal{U} \check{\mathcal{F}}$.

Proof. Fix $\mathcal{G} \in \mathbf{Set}^{\mathbf{L}_{\boldsymbol{\Sigma}}^{op}}$. By definition of $\mathcal{U}$, $\breve{\mathcal{F}}$ and $\widehat{\mathcal{F}}$ the following diagram commutes, where $\iota \colon |\mathbf{L}_{\boldsymbol{\Sigma}}^{op}| \hookrightarrow \mathbf{L}_{\boldsymbol{\Sigma}}^{op}$ is the inclusion functor.

$$
\begin{array}{ccc}
& \xrightarrow{\widehat{\mathcal{F}}\mathcal{U}\mathcal{G}} & \\
|\mathbf{L}_{\boldsymbol{\Sigma}}^{op}| \xrightarrow{\;\mathcal{U}\mathcal{G}\;} & \mathbf{Set} \xrightarrow{\;\mathcal{F}\;} & \mathbf{Set} \\
\iota \downarrow & \nearrow \mathcal{G} & \\
\mathbf{L}_{\boldsymbol{\Sigma}}^{op} & \xrightarrow[\breve{\mathcal{F}}\mathcal{G}]{} &
\end{array}
$$

This gives the following derivation: $\widehat{\mathcal{F}}\mathcal{U}\mathcal{G} = \mathcal{F}\mathcal{U}\mathcal{G} = \mathcal{F}\mathcal{G}\iota = \breve{\mathcal{F}}\mathcal{G}\iota = \mathcal{U}\breve{\mathcal{F}}\mathcal{G}$. The reasoning on arrows of $\mathbf{Set}^{\mathbf{L}_{\boldsymbol{\Sigma}}^{op}}$ is analogous. $\qquad\square$

By Proposition 9.2, we can consider $p^{\sharp} \colon At \to \mathcal{K}\widehat{\mathcal{P}_c}\widehat{\mathcal{L}}\mathcal{U}(At)$ as having the type $p^{\sharp} \colon At \to \mathcal{K}\mathcal{U}\breve{\mathcal{P}}_c\breve{\mathcal{L}}(At)$. Thus the bialgebra constructed out of $p^{\sharp}$ will be formed by a $\mathcal{K}\mathcal{U}\breve{\mathcal{P}}_c$-coalgebra $p_{\delta}^{\sharp} \colon \breve{\mathcal{L}}At \to \mathcal{K}\mathcal{U}\breve{\mathcal{P}}_c\breve{\mathcal{L}}(At)$. In order to define it by applying Proposition 2.3, the next step is to define δ as a distributive law of type $\breve{\mathcal{L}}(\mathcal{K}\mathcal{U}\breve{\mathcal{P}}_c) \Rightarrow (\mathcal{K}\mathcal{U}\breve{\mathcal{P}}_c)\breve{\mathcal{L}}$.

For this purpose, our strategy will be to construct δ as the combination of different distributive laws. First, recall the distributive law $\lambda \colon \mathcal{L}\mathcal{P}_f \Rightarrow \mathcal{P}_f\mathcal{L}$ of monads introduced in Section 7: we override notation by calling λ the distributive law of type $\mathcal{L}\mathcal{P}_c \Rightarrow \mathcal{P}_c\mathcal{L}$ defined as the one involving $\mathcal{P}_f$. By Proposition 2.2, $\lambda \colon \mathcal{L}\mathcal{P}_c \Rightarrow \mathcal{P}_c\mathcal{L}$ extends to a distributive law of monads $\breve{\lambda} \colon \breve{\mathcal{L}}\breve{\mathcal{P}}_c \Rightarrow \breve{\mathcal{P}}_c\breve{\mathcal{L}}$ in $\mathbf{Set}^{\mathbf{L}_{\boldsymbol{\Sigma}}^{op}}$. Next, we introduce two other distributive laws: one for $\breve{\mathcal{P}}_c$ over $\mathcal{K}\mathcal{U}$ and the other for $\breve{\mathcal{L}}$ over $\mathcal{K}\mathcal{U}$. In fact, because $\mathcal{K}\mathcal{U}$ is a monad arising from the adjunction $\mathcal{U} \dashv \mathcal{K}$ and extensions commute with $\mathcal{U}$ (Proposition 9.2), we can let

$$
\begin{aligned}
\varphi \colon & \quad \breve{\mathcal{L}}(\mathcal{K}\mathcal{U}) \;\Rightarrow\; (\mathcal{K}\mathcal{U})\breve{\mathcal{L}} \\
\psi \colon & \quad \breve{\mathcal{P}}_c(\mathcal{K}\mathcal{U}) \;\Rightarrow\; (\mathcal{K}\mathcal{U})\breve{\mathcal{P}}_c
\end{aligned}
$$

be defined in a canonical way, using the following general result.

Proposition 9.3. *Let* $\mathbf{C}$ *and* $\mathbf{D}$ *be categories with an adjunction* $\mathcal{U} \dashv \mathcal{K}$ *for* $\mathcal{U} \colon \mathbf{C} \to \mathbf{D}$ *and* $\mathcal{K} \colon \mathbf{D} \to \mathbf{C}$. *Let* $\breve{\mathcal{T}} \colon \mathbf{C} \to \mathbf{C}$ *and* $\widehat{\mathcal{T}} \colon \mathbf{D} \to \mathbf{D}$ *be two monads such that* $\mathcal{U}\breve{\mathcal{T}} = \widehat{\mathcal{T}}\mathcal{U}$. *Then, there is a distributive law of monads* $\lambda \colon \breve{\mathcal{T}}(\mathcal{K}\mathcal{U}) \Rightarrow (\mathcal{K}\mathcal{U})\breve{\mathcal{T}}$ *defined for all* $X \in \mathbf{C}$ *by*

$$
(\widehat{\mathcal{T}}(id_{\mathcal{K}\mathcal{U}X})^{\flat})^{\sharp} \tag{9.1}
$$

where $(\cdot)^{\flat}_{X,Z} \colon \mathbf{C}[X, \mathcal{K}Z] \to \mathbf{D}[\mathcal{U}X, Z]$ *and* $(\cdot)^{\sharp}_{X,Z} \colon \mathbf{D}[\mathcal{U}X, Z] \to \mathbf{C}[X, \mathcal{K}Z]$ *are the components of the canonical bijection given by the adjunction* $\mathcal{U} \dashv \mathcal{K}$.

Proof. See Appendix A. $\qquad\square$

In our case, $\mathbf{C} = \mathbf{Set}^{\mathbf{L}_{\boldsymbol{\Sigma}}^{op}}$, $\mathbf{D} = \mathbf{Set}^{|\mathbf{L}_{\boldsymbol{\Sigma}}^{op}|}$ and the required property of commuting with $\mathcal{U}$ is given for both pairs of monads $\breve{\mathcal{L}}, \widehat{\mathcal{L}}$ and $\breve{\mathcal{P}}_c, \widehat{\mathcal{P}_c}$ by Proposition 9.2. In the sequel we provide the explicit calculation of the distributive law $\varphi \colon \breve{\mathcal{L}}\mathcal{K}\mathcal{U} \Rightarrow \mathcal{K}\mathcal{U}\breve{\mathcal{L}}$ according to (9.1). For this purpose, fix a presheaf $\mathcal{G} \in \mathbf{Set}^{\mathbf{L}_{\boldsymbol{\Sigma}}^{op}}$ and $n \in \mathbf{L}_{\boldsymbol{\Sigma}}^{op}$. By definition $\varphi_{\mathcal{G}} \colon \mathbf{Set}^{\mathbf{L}_{\boldsymbol{\Sigma}}^{op}} \to \mathbf{Set}^{\mathbf{L}_{\boldsymbol{\Sigma}}^{op}}$ is given on $n \in \mathbf{L}_{\boldsymbol{\Sigma}}^{op}$ as the following function (where $\mathcal{U}\breve{\mathcal{L}} = \widehat{\mathcal{L}}\mathcal{U}$ by Proposition 9.2):

$$
(\widehat{\mathcal{L}}(id_{\mathcal{K}\mathcal{U}\mathcal{G}})^{\flat})^{\sharp}(n) \colon \breve{\mathcal{L}}\mathcal{K}\mathcal{U}\mathcal{G}(n) \xrightarrow{\;\eta_{\breve{\mathcal{L}}\mathcal{K}\mathcal{U}\mathcal{G}}(n)\;} \mathcal{K}\mathcal{U}\breve{\mathcal{L}}\mathcal{K}\mathcal{U}\mathcal{G}(n) \xrightarrow{\;\mathcal{K}(\widehat{\mathcal{L}}(id_{\mathcal{K}\mathcal{U}\mathcal{G}})^{\flat})(n)\;} \mathcal{K}\mathcal{U}\breve{\mathcal{L}}\mathcal{G}(n).
$$

We now compute $\varphi_\mathcal{G}(n)$ on a list of tuples $[\dot{x}_1, \ldots, \dot{x}_k]$. This is first mapped onto the value

$$[\dot{x}_1, \ldots, \dot{x}_k] \xmapsto{\eta_{\check{\mathcal{L}}\mathcal{K}\mathcal{U}\mathcal{G}}(n)} \langle \check{\mathcal{L}}\mathcal{K}\mathcal{U}\mathcal{G}(\theta)[\dot{x}_1, \ldots, \dot{x}_k]\rangle_{\theta\colon n\to m} = \langle[\langle\dot{x}_1(\sigma\circ\theta)\rangle_\sigma, \ldots, \langle\dot{x}_k(\sigma\circ\theta)\rangle_\sigma]\rangle_\theta$$

where the unit η is computed as in (4.2) and the equality follows by definition of $\mathcal{K}(\mathcal{U}\mathcal{G})$ on arrows $\theta \in \mathbf{L}_{\boldsymbol{\Sigma}}^{op}[n, m]$ — cf. (4.1). Then we apply $(id_{\mathcal{K}\mathcal{U}\mathcal{G}})^\flat$ componentwise in the list elements of the tuple:

$$\langle[\langle\dot{x}_1(\sigma\circ\theta)\rangle_\sigma, \ldots, \langle\dot{x}_k(\sigma\circ\theta)\rangle_\sigma]\rangle_\theta \xmapsto{\mathcal{K}(\widehat{\mathcal{L}}(id_{\mathcal{K}\mathcal{U}\mathcal{G}})^\flat)(n)} \langle[\langle\dot{x}_1(\sigma\circ\theta)\rangle_\sigma(id_n), \ldots, \langle\dot{x}_k(\sigma\circ\theta)\rangle_\sigma(id_n)]\rangle_\theta$$
$$= \langle[\dot{x}_1(\theta), \ldots, \dot{x}_k(\theta)]\rangle_\theta.$$

By definition of $(\cdot)^\flat$, $(id_{\mathcal{K}\mathcal{U}\mathcal{G}})^\flat(n) = (\epsilon_{\mathcal{U}\mathcal{G}} \circ \mathcal{U}(id_{\mathcal{K}\mathcal{U}\mathcal{G}})(n)\colon \mathcal{U}\mathcal{K}\mathcal{U}\mathcal{G}(n) \to \mathcal{U}\mathcal{G}(n)$ maps a tuple $\dot{x}$ into its element $\dot{x}(id_n)$ — cf. the definition (5.1) of the counit ϵ. The equality holds because $\langle\dot{x}_i(\sigma\circ\theta)\rangle_\sigma(id_n) = \dot{x}_i(id_n\circ\theta) = \dot{x}_i(\theta)$.

The calculation leading to the definition of $\psi\colon \breve{\mathcal{P}}_c\mathcal{K}\mathcal{U} \Rightarrow \mathcal{K}\mathcal{U}\breve{\mathcal{P}}_c$ is analogous. In conclusion we obtain the following definitions for distributive laws of monads φ and ψ:

$$\begin{aligned}
\varphi_\mathcal{G}(n)\colon \quad &\check{\mathcal{L}}\mathcal{K}\mathcal{U}\mathcal{G}(n) &\to& \quad \mathcal{K}\mathcal{U}\check{\mathcal{L}}\mathcal{G}(n) \\
&[\dot{x}_1, \ldots, \dot{x}_k] &\mapsto& \quad \langle[\dot{x}_1(\theta), \ldots, \dot{x}_k(\theta)]\rangle_{\theta\colon n\to m} \\
\psi_\mathcal{G}(n)\colon \quad &\breve{\mathcal{P}}_c\mathcal{K}\mathcal{U}\mathcal{G}(n) &\to& \quad \mathcal{K}\mathcal{U}\breve{\mathcal{P}}_c\mathcal{G}(n) \\
&\{\dot{x}_i\}_{i\in I} &\mapsto& \quad \langle\{\dot{x}_i(\theta)\}_{i\in I}\rangle_{\theta\colon n\to m}
\end{aligned}$$

where I is a countable set of indices. Note that the existence of distributive laws φ and ψ implies in particular that $\mathcal{K}\mathcal{U}\check{\mathcal{L}}$ and $\mathcal{K}\mathcal{U}\breve{\mathcal{P}}_c$ are monads.

We have now all ingredients to define the distributive law $\delta\colon \check{\mathcal{L}}(\mathcal{K}\mathcal{U}\breve{\mathcal{P}}_c) \Rightarrow (\mathcal{K}\mathcal{U}\breve{\mathcal{P}}_c)\check{\mathcal{L}}$:

$$\delta\colon \qquad \check{\mathcal{L}}\mathcal{K}\mathcal{U}\breve{\mathcal{P}}_c \xrightarrow{\varphi_{\breve{\mathcal{P}}_c}} \mathcal{K}\mathcal{U}\check{\mathcal{L}}\breve{\mathcal{P}}_c \xrightarrow{\mathcal{K}\mathcal{U}\check{\lambda}} \mathcal{K}\mathcal{U}\breve{\mathcal{P}}_c\check{\mathcal{L}} \qquad (9.2)$$

Concretely, given $\mathcal{G} \in \mathbf{Set}^{\mathbf{L}_{\boldsymbol{\Sigma}}^{op}}$ and $n \in \mathbf{L}_{\boldsymbol{\Sigma}}^{op}$, the function $\delta_\mathcal{G}(n)$ is defined by

$$[\langle X_1\rangle_\theta, \ldots, \langle X_k\rangle_\theta] \xmapsto{\varphi_{\breve{\mathcal{P}}_c\mathcal{G}}(n)} \langle[X_1, \ldots, X_k]\rangle_\theta \xmapsto{\mathcal{K}\mathcal{U}\check{\lambda}_\mathcal{G}(n)} \langle\{[x_1, \ldots, x_k] \mid x_i \in X_i\}\rangle_\theta$$

Proposition 9.4. $\delta\colon \check{\mathcal{L}}(\mathcal{K}\mathcal{U}\breve{\mathcal{P}}_c) \Rightarrow (\mathcal{K}\mathcal{U}\breve{\mathcal{P}}_c)\check{\mathcal{L}}$ *is a distributive law of the monad* $\check{\mathcal{L}}$ *over the monad* $\mathcal{K}\mathcal{U}\breve{\mathcal{P}}_c$.

Proof. In [14] it is proven that the natural transformation δ defined as in (9.2) (or, equivalently, the natural transformation $\psi_{\check{\mathcal{L}}} \circ \breve{\mathcal{P}}_c\varphi\colon \breve{\mathcal{P}}_c\check{\mathcal{L}}\mathcal{K}\mathcal{U} \Rightarrow \mathcal{K}\mathcal{U}\breve{\mathcal{P}}_c\check{\mathcal{L}}$) is a distributive law yielding the monad $\mathcal{K}\mathcal{U}\breve{\mathcal{P}}_c\check{\mathcal{L}}$ if one can prove that the three distributive laws $\check{\lambda}$, φ and ψ satisfy a compatibility condition called Yang-Baxter equation. This is given by commutativity of the following diagram, which can be easily verified by definition of $\check{\lambda}$, φ and ψ.

$\square$

Convention 9.5. Throughout the rest of the paper, we do not need to manipulate further the components of the functor $\mathcal{K}\mathcal{U}\breve{\mathcal{P}}_c \colon \mathbf{Set}^{\mathbf{L}_\Sigma^{op}} \to \mathbf{Set}^{\mathbf{L}_\Sigma^{op}}$ and thus we adopt the shorter notation $\mathcal{R}$ for it.

We are now in position to extend Construction 7.3 to arbitrary logic programs.

Construction 9.6. Let $\mathbb{P}$ be a logic program and $p^\sharp \colon At \to \mathcal{R}\breve{\mathcal{L}}(At)$ be the associated coalgebra in $\mathbf{Set}^{\mathbf{L}_\Sigma^{op}}$. We define the $\mathcal{R}$-coalgebra $p_\delta^\sharp \colon \mathcal{L}At \to \mathcal{R}\breve{\mathcal{L}}(At)$ in the way prescribed by Proposition 2.3 as

$$\breve{\mathcal{L}}At \xrightarrow{\breve{\mathcal{L}}p^\sharp} \breve{\mathcal{L}}\mathcal{R}\breve{\mathcal{L}}At \xrightarrow{\delta_{\breve{\mathcal{L}}At}} \mathcal{R}\breve{\mathcal{L}}\breve{\mathcal{L}}At \xrightarrow{\mathcal{R}(::)} \mathcal{R}\breve{\mathcal{L}}At$$

where $::$ is $\mu_{At}^{\breve{\mathcal{L}}}$, defined componentwise by list concatenation $\mu^{\mathcal{L}}$ since $\breve{\mathcal{L}}$ is the extension of $\mathcal{L}$. By Proposition 2.3, $(\mathcal{L}At, ::, p_\delta^\sharp)$ forms a δ-bialgebra.

In order to have a more concrete intuition, we spell out the details of the above construction. Fixed $n \in \mathbf{L}_\Sigma^{op}$, $\breve{\mathcal{L}}p^\sharp(n)$ maps a list $[A_1, \ldots A_k] \in \mathcal{L}At(n)$ into the list of tuples

$$[\langle X_1 \rangle_\theta, \ldots \langle X_k \rangle_\theta]$$

where each $X_i = p^\sharp(n)(A_i)(\theta)$ is a set of lists of atoms. This is mapped by $\delta_{\breve{\mathcal{L}}At}$ into the tuple of sets (of lists of lists)

$$\langle \{[l_1, \ldots, l_k] | l_i \in X_i\} \rangle_\theta$$

and, finally, by $\mathcal{R}(::)$ into the tuple of sets (of list)

$$\langle \{l_1 :: \ldots :: l_k | l_i \in X_i\} \rangle_\theta.$$

Alternatively, $p_\delta^\sharp$ can be inductively defined from $p^\sharp$ by the following rules, where $l \xrightarrow{\theta} l'$ stands for $l' \in p_\delta^\sharp(l)(\theta)$.

$$l1 \quad \frac{l' \in p^\sharp(A)(\theta)}{[A] \xrightarrow{\theta} l'} \qquad l2 \quad \frac{l_1 \xrightarrow{\theta} l_1' \quad l_2 \xrightarrow{\theta} l_2'}{l_1 :: l_2 \xrightarrow{\theta} l_1' :: l_2'} \qquad l3 \quad \frac{}{[\,] \xrightarrow{\theta} [\,]}$$

The rule system extends the one provided for the ground case (*cf.* Remark 7.6) by labeling transitions with the substitution applied on the goal side. Observe that rule ($l2$) is the same as the one for parallel composition in CSP [44]: the composite system can evolve only if its parallel components are able to synchronise on some common label θ.

Example 9.7. Consider the logic program $\mathtt{NatList}$ in Example 3.6 and the atoms $\mathtt{Nat}(x_1)$ and $\mathtt{List}(cons(x_1, x_2))$, both in $At(2)$. The morphism $p^\sharp \colon At \to \mathcal{K}\mathcal{U}\breve{\mathcal{P}}_c\breve{\mathcal{L}}(At)$ maps these atoms into the tuples defined for all $\theta \in \mathbf{L}_\Sigma^{op}[2, m]$ as

$$p^\sharp(\mathtt{Nat}(x_1))(\theta) \;=\; \begin{cases} \{[\,]\} & \text{if } x_1\theta = zero \\ \{[\mathtt{Nat}(t)]\} & \text{if } x_1\theta = succ(t) \\ \emptyset & \text{otherwise} \end{cases}$$

$$p^\sharp(\mathtt{List}(cons(x_1, x_2)))(\theta) \;=\; \{[\mathtt{Nat}(x_1)\theta, \mathtt{List}(x_2)\theta]\}$$

for some Σ-term t. By application of rule ($l1$), such tuples are the same of $p_\delta^\sharp([\mathtt{Nat}(x_1)])$ and $p_\delta^\sharp([\mathtt{List}(cons(x_1, x_2))])$ and, by mean of rule ($l2$), it is easy to compute the value of $p_\delta^\sharp$

on the list of atoms $[\mathtt{Nat}(x_1), \mathtt{List}(cons(x_1, x_2))]$.

$$p_\delta^\sharp([\mathtt{Nat}(x_1), \mathtt{List}(cons(x_1, x_2))])(\theta) = \begin{cases} \{[\mathtt{Nat}(zero), \mathtt{List}(x_2)\theta]\} & \text{if } x_1\theta = zero \\ \{[\mathtt{Nat}(t), \mathtt{Nat}(succ(t)), \mathtt{List}(x_2)\theta]\} & \text{if } x_1\theta = succ(t) \\ \emptyset & \text{otherwise} \end{cases}$$

Intuitively, $p_\delta^\sharp$ forces all the atoms of a list to synchronize by choosing a common substitution. For instance, $[\mathtt{List}(cons(x_1, x_2))]$ can make a transition with any substitution θ but, when in parallel with $[\mathtt{Nat}(x_1)]$, it cannot evolve (and thus cannot be refuted) for those substitutions that do not allow $\mathtt{Nat}(x_1)$ to evolve — i.e., those θ belonging to the third case above.

We now generalize Construction 7.7 to define the cofree semantics $\llbracket \cdot \rrbracket_{p_\delta^\sharp}$ arising from $p_\delta^\sharp$.

Construction 9.8. The cofree $\mathcal{R}$-coalgebra $\mathcal{C}(\mathcal{R})(\check{\mathcal{L}}At)$ on $\check{\mathcal{L}}At$, defined following the same steps of Construction 4.3, forms the final $\check{\mathcal{L}}At \times \mathcal{R}(\cdot)$-coalgebra $c\colon \mathcal{C}(\mathcal{R})(\check{\mathcal{L}}At) \xrightarrow{\cong} \check{\mathcal{L}}At \times \mathcal{R}(\mathcal{C}(\mathcal{R})(\check{\mathcal{L}}At))$. We now build its canonical extension to a final δ'-bialgebra, where $\delta'\colon \check{\mathcal{L}}(\check{\mathcal{L}}At \times \mathcal{R}(\cdot)) \Rightarrow \check{\mathcal{L}}At \times \mathcal{R}\check{\mathcal{L}}(\cdot)$ is a distributive law of the monad $\check{\mathcal{L}}$ over the functor $\check{\mathcal{L}}At \times \mathcal{R}(\cdot)$ defined in terms of δ:

$$\delta'_{\mathcal{G}}\colon \ \check{\mathcal{L}}(\check{\mathcal{L}}At \times \mathcal{R}(\mathcal{G})) \xrightarrow{\langle \check{\mathcal{L}}\pi_1, \check{\mathcal{L}}\pi_2 \rangle} \check{\mathcal{L}}\check{\mathcal{L}}At \times \check{\mathcal{L}}\mathcal{R}(\mathcal{G}) \xrightarrow{::\times \delta_{\mathcal{G}}} \check{\mathcal{L}}At \times \mathcal{R}\check{\mathcal{L}}(\mathcal{G}) \ .$$

For this purpose, we construct a $\check{\mathcal{L}}$-algebra $\bar{::}\colon \check{\mathcal{L}}(\mathcal{C}(\mathcal{R})(\check{\mathcal{L}}At)) \to \mathcal{C}(\mathcal{R})(\check{\mathcal{L}}At)$, using finality of $\mathcal{C}(\mathcal{R})(\check{\mathcal{L}}At)$:

$$
\begin{array}{ccc}
\check{\mathcal{L}}(\mathcal{C}(\mathcal{R})(\check{\mathcal{L}}At)) & \dashrightarrow^{\bar{\bar{::}}} & \mathcal{C}(\mathcal{R})(\check{\mathcal{L}}At) \\
{\scriptstyle \check{\mathcal{L}}c} \downarrow & & \downarrow {\scriptstyle c} \\
\check{\mathcal{L}}(\check{\mathcal{L}}At \times \mathcal{R}(\mathcal{C}(\mathcal{R})(\check{\mathcal{L}}At))) & & \\
{\scriptstyle \delta'_{\mathcal{C}(\mathcal{R})(\check{\mathcal{L}}At)}} \downarrow & & \\
\check{\mathcal{L}}At \times \mathcal{R}(\check{\mathcal{L}}(\mathcal{C}(\mathcal{R})(\check{\mathcal{L}}At))) & \xrightarrow{id_{\check{\mathcal{L}}At} \times \mathcal{R}(\bar{::})} & \check{\mathcal{L}}At \times \mathcal{R}(\mathcal{C}(\mathcal{R})(\check{\mathcal{L}}At))
\end{array}
$$

Proposition 2.4 guarantees that $\big(\mathcal{C}(\mathcal{R})(\check{\mathcal{L}}At), \bar{::}, c\big)$ is the final δ'-bialgebra.

We now turn to the definition of the semantics $\llbracket \cdot \rrbracket_{p_\delta^\sharp}\colon \check{\mathcal{L}}At \to \mathcal{C}(\mathcal{R})(\check{\mathcal{L}}At)$. We let it be the unique δ'-bialgebra morphism from $\check{\mathcal{L}}At$ to $\mathcal{C}(\mathcal{R})(\check{\mathcal{L}}At)$, given by finality of $\big(\mathcal{C}(\mathcal{R})(\check{\mathcal{L}}At), \bar{::}, c\big)$:

$$
\begin{array}{ccc}
\check{\mathcal{L}}\check{\mathcal{L}}At & \xrightarrow{\check{\mathcal{L}}\llbracket \cdot \rrbracket_{p_\delta^\sharp}} & \check{\mathcal{L}}(\mathcal{C}(\mathcal{R})(\check{\mathcal{L}}At)) \\
{\scriptstyle ::} \downarrow & & \downarrow {\scriptstyle \bar{::}} \\
\check{\mathcal{L}}At & \dashrightarrow^{\llbracket \cdot \rrbracket_{p_\delta^\sharp}} & \mathcal{C}(\mathcal{R})(\check{\mathcal{L}}At) \\
{\scriptstyle \langle id_{\check{\mathcal{L}}At}, p_\delta^\sharp \rangle} \downarrow & & \downarrow {\scriptstyle c} \\
\check{\mathcal{L}}At \times \mathcal{R}(\check{\mathcal{L}}At) & \xrightarrow{id_{\check{\mathcal{L}}At} \times \mathcal{R}(\llbracket \cdot \rrbracket_{p_\delta^\sharp})} & \check{\mathcal{L}}At \times \mathcal{R}(\mathcal{C}(\mathcal{R})(\check{\mathcal{L}}At)).
\end{array}
$$

The next result states that bialgebraic semantics exhibits two forms of compositionality: it respects both the substitutions in $\mathbf{L}_\Sigma^{op}$ (by saturation) and the internal structure of goals.
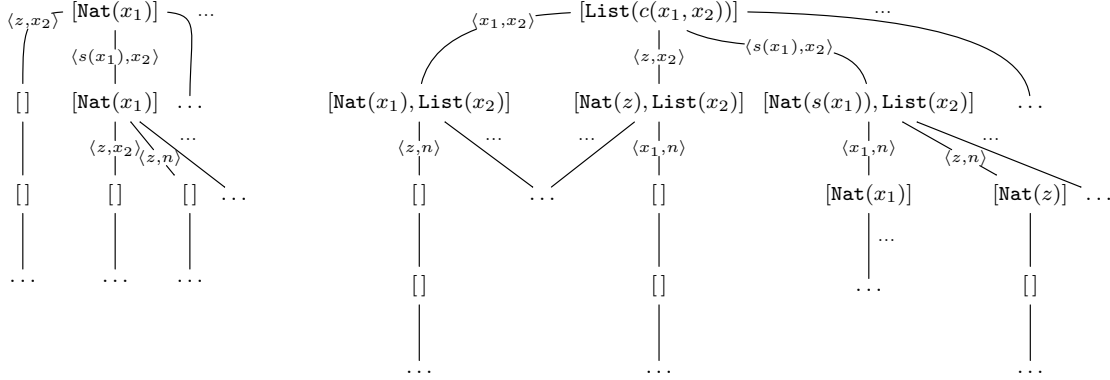
Figure 2: Part of the bialgebraic semantics of $[\mathtt{Nat}(x_1)]$ (left) and $[\mathtt{List}(cons(x_1, x_2))]$ (right) in $At(2)$. We use the convention that unlabeled edges stand for edges with any substitution.
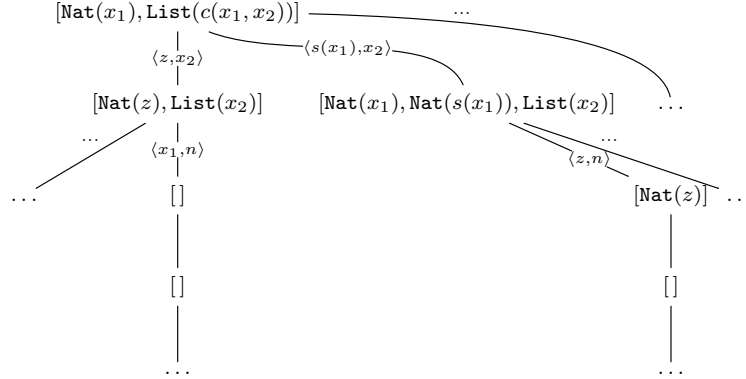

Figure 3: Part of the bialgebraic semantics of $[\mathtt{Nat}(x_1), \mathtt{List}(cons(x_1, x_2))]$.

In order to formulate such theorem, given a substitution $\theta \in \mathbf{L}_{\boldsymbol{\Sigma}}^{op}[n, m]$, we use notation:

$$
\begin{aligned}
\theta^l &\coloneqq \quad \check{\mathcal{L}}(At)(\theta)\colon \check{\mathcal{L}}(At)(n) \to \check{\mathcal{L}}(At)(m) \\
\overline{\theta} &\coloneqq \quad \mathcal{C}(\mathcal{R})(\mathcal{L}At)(\theta)\colon \mathcal{C}(\mathcal{R})(\mathcal{L}At)(n) \to \mathcal{C}(\mathcal{R})(\mathcal{L}At)(m).
\end{aligned}
$$

Theorem 9.9 (Two-Fold Compositionality)**.** *Let* $l_1, \dots, l_k$ *be list of atoms in* $At(n)$ *and* $\theta \in \mathbf{L}_{\boldsymbol{\Sigma}}^{op}[n, m]$ *a substitution. The following two equalities hold:*

$$
[\![\theta^l\, l_1 :: \dots :: \theta^l\, l_k]\!]_{p_{\delta}^{\sharp}} = \overline{\theta}[\![l_1]\!]_{p_{\delta}^{\sharp}} \overline{::} \dots \overline{::} \overline{\theta}[\![l_k]\!]_{p_{\delta}^{\sharp}} = \overline{\theta}([\![l_1]\!]_{p_{\delta}^{\sharp}} \overline{::} \dots \overline{::} [\![l_k]\!]_{p_{\delta}^{\sharp}}).
$$

Where $l_1 :: \dots :: l_k$ *is notation for* $::([l_1, \dots, l_k]) = \mu_{At}^{\check{\mathcal{L}}}(n)([l_1, \dots, l_k])$ *and* $[\![l_1]\!]_{p_{\delta}^{\sharp}} \overline{::} \dots \overline{::} [\![l_k]\!]_{p_{\delta}^{\sharp}}$ *for* $\overline{::}(n)([\![l_1]\!]_{p_{\delta}^{\sharp}}, \dots, [\![l_k]\!]_{p_{\delta}^{\sharp}}).$

Proof. The proof is entirely analogous to the one for the ground case, see Theorem 7.8. Commutativity with substitutions is given by naturality of $[\![\cdot]\!]_{p_{\delta}^{\sharp}}$, $::$ and $\overline{::}$ in $\mathbf{Set}^{\mathbf{L}_{\boldsymbol{\Sigma}}^{op}}$. $\square$

Example 9.10. In Example 9.7 we have computed the values of $p_\delta^\sharp$ for the lists $[\mathtt{Nat}(x_1)]$ and $[\mathtt{List}(cons(x_1, x_2))]$. Figure 2 shows (a finite part of) their bialgebraic semantics $[\![\mathtt{Nat}(x_1)]\!]_{p_\delta^\sharp}$ and $[\![\mathtt{List}(cons(x_1, x_2))]\!]_{p_\delta^\sharp}$. These are depicted as $\vee$-trees (Definition 8.1) where edges are labeled with substitutions. Analogously to the ground case, one can think of the edges as (labeled) transitions generated by the rule presentation of p_δ given above.

It is instructive to observe that, while $[\mathtt{Nat}(x_1)]$ has one $\langle zero, x_2 \rangle$-child, $[\mathtt{Nat}(x_1), \mathtt{List}(x_2)]$ cannot have a child with such substitution: in order to progress $[\mathtt{Nat}(x_1), \mathtt{List}(x_2)]$ needs a substitution which makes progress at the same time both $\mathtt{Nat}(x_1)$ and $\mathtt{List}(x_2)$ like, for instance, $\langle zero, nil \rangle$.

In Example 9.7 we discussed the value $p_\delta^\sharp([\mathtt{Nat}(x_1), \mathtt{List}(cons(x_1, x_2))])$. Figure 3 shows the bialgebraic semantics $[\![\mathtt{Nat}(x_1), \mathtt{List}(cons(x_1, x_2))]\!]_{p_\delta^\sharp}$ for such list. By virtue of Theorem 9.9, it can be equivalently obtained by concatenating via $\overline{::}$ the trees $[\![\mathtt{Nat}(x_1)]\!]_{p_\delta^\sharp}$ and $[\![\mathtt{List}(cons(x_1, x_2))]\!]_{p_\delta^\sharp}$ in Figure 2. Similarly to the ground case, the operation of concatenating two trees T_1, T_2 can be described as follows.

(1) If the root of T_1 has label l_1 and the root of T_2 has label l_2, then the root of $T_1 \overline{::} T_2$ has label $l_1 :: l_2$;
(2) if T_1 has a θ-child T_1' and T_2 has a θ-child T_2', then $T_1 \overline{::} T_2$ has a θ-child $T_1' \overline{::} T_2'$.

For instance, while $[\mathtt{List}(cons(x_1, x_2))]$ has one $\langle x_1, x_2 \rangle$-child, $[\mathtt{Nat}(x_1), \mathtt{List}(cons(x_1, x_2))]$ has no $\langle x_1, x_2 \rangle$-children because $[\mathtt{Nat}(x_1)]$ has no $\langle x_1, x_2 \rangle$-children. Instead it has one $\langle zero, x_2 \rangle$-child labeled with $[\,] :: [\mathtt{Nat}(zero), \mathtt{List}(x_2)]$ and one $\langle succ(x_1), x_2 \rangle$-child labeled with $[\mathtt{Nat}(x_1)] :: [\mathtt{Nat}(succ(x_1)), \mathtt{List}(x_2)]$. The latter node has no $\langle x_1, nil \rangle$-children because $[\mathtt{Nat}(x_1)]$ has no $\langle x_1, nil \rangle$-children.

10. Soundness and Completeness of Bialgebraic Semantics

In this section we study the relationship between the bialgebraic semantics $[\![\cdot]\!]_{p_\delta^\sharp}$ and the other approaches investigated so far. First, analogously to the ground case, we provide an explicit description of the elements of $\mathcal{C}(\mathcal{P}_f)(\mathcal{L}At)$ that $[\![\cdot]\!]_{p_\delta^\sharp}$ associates with goals. This formalizes the notion of tree given in Example 9.10.

Definition 10.1. Given a logic program $\mathbb{P}$, $n \in \mathbf{L}_{\boldsymbol{\Sigma}}^{op}$ and a list $l \in \mathcal{L}At(n)$ of atoms, the *(parallel) saturated $\vee$-tree* for l in $\mathbb{P}$ is the possibly infinite tree T satisfying the following properties:

(1) Each node s in T is labeled with a list of atoms $l_s \in \mathcal{L}At(m)$ for some $m \in \mathbf{L}_{\boldsymbol{\Sigma}}^{op}$ and the root is labeled with l. For any child t of s, say labeled with a list $l_t \in \mathcal{L}At(z)$, the edge from s to t is labeled with a substitution $\sigma \colon m \to z$.
(2) Let s be a node in T with label $l' = [A_1, \ldots, A_k] \in \mathcal{L}At(m)$. For all substitutions $\sigma, \tau_1, \ldots, \tau_k$ and list $[C_1, \ldots, C_k]$ of clauses of $\mathbb{P}$ such that, for each $C_i = H^i \leftarrow B_1^i, \ldots, B_j^i$, $\langle \sigma, \tau_i \rangle$ is a unifier of A_i and H^i, s has exactly one child t, and viceversa. Furthermore, the edge connecting s to t is labeled with σ and the node t is labeled with the list $l_1 :: \ldots :: l_k$, where $l_i = [\tau_i B_1^i, \ldots, \tau_i B_j^i]$ is given by applying τ_i to the body of clause C_i.

Saturated $\vee$-trees extend the $\vee$-trees introduced in Definition 8.1 and we can formulate the same adequacy result.

Proposition 10.2 (Adequacy). *Given a list of atoms $l \in \mathcal{L}At(n)$ and a program $\mathbb{P}$, $[\![l]\!]_{p_\delta^\sharp} \in \mathcal{C}(\mathcal{P}_f)(\mathcal{L}At)(n)$ is the saturated $\vee$-tree for l in $\mathbb{P}$.*

It is worth clarifying that the substitutions labeling edges in a saturated $\vee$-tree play the same role as the substitutions labeling or-nodes in saturated $\wedge\vee$-trees. For any node in $[\![l]\!]_{p_\delta^\sharp}$, say labeled with l_1, an outgoing edge with label θ connecting to a child l_2 means that l_2 is an element of $p_\delta^\sharp(l')(\theta)$ — *cf.* the rule presentation of $p_\delta^\sharp$ in Section 9.

Analogously to Construction 8.3, we now set a translation $r_{sat} \colon \mathcal{C}(\mathcal{R}\check{\mathcal{L}})(At) \to \mathcal{C}(\mathcal{R})(\check{\mathcal{L}}At)$ between saturated $\vee\wedge$-trees (Definition 4.5) and saturated $\vee$-trees.

Construction 10.3. Let $u \colon \mathcal{C}(\mathcal{R}\check{\mathcal{L}})(At) \xrightarrow{\cong} \mathcal{R}\check{\mathcal{L}}At \times \big(\mathcal{C}(\mathcal{R}\check{\mathcal{L}})(At)\big)$ be the final $At \times \mathcal{R}\check{\mathcal{L}}(\cdot)$-coalgebra structure on $\mathcal{C}(\mathcal{R}\check{\mathcal{L}})(At)$ and define $d := \eta_{At}^{\check{\mathcal{L}}} \times id \circ u$ as the extension of u to a $\check{\mathcal{L}}At \times \mathcal{R}\check{\mathcal{L}}(\cdot)$-coalgebra. Next we provide a $\check{\mathcal{L}}At \times \mathcal{R}(\cdot)$-coalgebra structure $d_{\delta'}$ on $\check{\mathcal{L}}(\mathcal{C}(\mathcal{R}\check{\mathcal{L}})(At))$ in the way prescribed by Proposition 2.3:

$$
\begin{array}{ccc}
\check{\mathcal{L}}(\mathcal{C}(\mathcal{R}\check{\mathcal{L}})(At)) & \xrightarrow{\quad d_{\delta'} \quad} & \check{\mathcal{L}}At \times \mathcal{R}\check{\mathcal{L}}\big(\mathcal{C}(\mathcal{R}\check{\mathcal{L}})(At)\big) \\
{\scriptstyle \check{\mathcal{L}}d} \downarrow & \overset{\parallel}{\cdots} & \uparrow {\scriptstyle id_{\check{\mathcal{L}}At} \times \mathcal{R}(::)} \\
\check{\mathcal{L}}\big(\check{\mathcal{L}}At \times \mathcal{R}\check{\mathcal{L}}\big(\mathcal{C}(\mathcal{R}\check{\mathcal{L}})(At)\big)\big) & \xrightarrow{\lambda'_{\mathcal{C}(\mathcal{R}\check{\mathcal{L}})(At)}} & \check{\mathcal{L}}At \times \mathcal{R}\check{\mathcal{L}}\check{\mathcal{L}}\big(\mathcal{C}(\mathcal{R}\check{\mathcal{L}})(At)\big)
\end{array}
$$

By finality of $\mathcal{C}(\mathcal{R})(\check{\mathcal{L}}At)$ we obtain our representation map $r_{sat} := [\![\cdot]\!]_{d_{\delta'}} \circ \eta_{\mathcal{C}(\mathcal{R}\check{\mathcal{L}})(At)}^{\check{\mathcal{L}}}$.

$$
\begin{array}{ccccc}
\mathcal{C}(\mathcal{R}\check{\mathcal{L}})(At) & \xrightarrow{\eta_{\mathcal{C}(\mathcal{R}\check{\mathcal{L}})(At)}^{\check{\mathcal{L}}}} & \check{\mathcal{L}}(\mathcal{C}(\mathcal{R}\check{\mathcal{L}})(At)) & \dashrightarrow^{[\![\cdot]\!]_{d_{\delta'}}} & \mathcal{C}(\mathcal{R})(\check{\mathcal{L}}At) \\
 & {\scriptstyle u} \downarrow & & & \\
d\big(& At \times \mathcal{R}\check{\mathcal{L}}\big(\mathcal{C}(\mathcal{R}\check{\mathcal{L}})(At)\big) & & & {\scriptstyle c} \\
 & {\scriptstyle \eta_{At}^{\check{\mathcal{L}}} \times id} \downarrow & {\scriptstyle d_{\delta'}} & & \\
\check{\mathcal{L}}At \times \mathcal{R}\check{\mathcal{L}}\big(\mathcal{C}(\mathcal{R}\check{\mathcal{L}})(At)\big) & \xrightarrow{\quad id_{\check{\mathcal{L}}At} \times \mathcal{R}([\![\cdot]\!]_{d_{\delta'}}) \quad} & & & \check{\mathcal{L}}At \times \mathcal{R}\mathcal{C}(\mathcal{R})(\check{\mathcal{L}}At)
\end{array}
\qquad (10.1)
$$

Proposition 10.4. *For any atom $A \in At(n)$, $r_{sat}([\![A]\!]_{p^\sharp}) = [\![A]\!]_{p_\delta^\sharp}$.*

Proof. The proof is entirely analogous to the one provided for the ground case, see Proposition 8.4. $\qquad\square$

We now focus on the notion of refutation subtree associated with saturated $\vee$-trees. As outlined in the introduction, here lies one of the main motivations for bialgebraic semantics. When defining refutation subtrees for saturated semantics (Definition 6.1), we had to require that they were *synched*. The corresponding operational intuition is that, at each derivation step, the proof-search for the atoms in the current goal can only advance by applying to all of them the same substitution. If we did not impose such condition, we would take into account derivation subtrees yielding an unsound refutation, where the same variable is substituted for different values, as shown in Example 3.4.

The deep reason for requiring such constraint is that, to be sound, the explicit and-parallelism exhibited by saturated $\wedge\vee$-trees has to respect some form of dependency between

the substitutions applied on different branches. Coinductive trees (Definition 3.5) achieve it by construction, because all substitutions applied in the goal have to be identities. For saturated ∨-trees, this property is also given by construction, but in a more general way: at each step the same substitution (not necessarily the identity) is applied on all the atoms of the goal. This synchronicity property is already encoded in the operational semantics $p_\delta^\sharp$, as immediately observable in its rule presentation, and arises by definition of δ.

By these considerations, subtrees of saturated ∨-trees are always synched (in the sense of Definition 6.1) and we can define a sound notion of derivation as in the ground case, without the need of additional constraints.

Definition 10.5. Let T be a saturated ∨-tree in $\mathcal{C}(\mathcal{R})(\mathcal{L}At)(n)$ for some $n \in \mathbf{L}_{\mathbf{\Sigma}}^{op}$. A *derivation subtree* of T is a sequence of nodes $s_1, s_2, \ldots$ such that s_1 is the root of T and s_{i+1} is a child of s_i. A *refutation subtree* is a finite derivation subtree $s_1, \ldots, s_k$ where the last element s_k is labeled with the empty list. Its *answer* is the substitution $\theta_k \circ \ldots \theta_2 \circ \theta_1$, where θ_i is the substitution labeling the edge between s_i and s_{i+1}.

The following statement about refutation subtrees will be useful later.

Proposition 10.6. *Let $\mathbb{P}$ be a logic program and $l \in \mathcal{L}At(n)$ a list of atoms. If $[\![l]\!]_{p_\delta^\sharp}$ has a refutation subtree with answer $\theta\colon n \to m$, then it has also a refutation subtree $T = s_1, s_2, \ldots$ with the same answer where the edge connecting s_1 and s_2 is labeled with θ and all the other edges in T are labeled with id_m.*

Proof. See Appendix A. □

Example 10.7. The saturated ∨-trees for $[\mathtt{Nat}(x_1)]$ and $[\mathtt{List}(cons(x_1, x_2))]$ in Figure 2 contain several refutation subtrees: for instance,

$$[\mathtt{List}(cons(x_1, x_2))] \xrightarrow{\langle zero, x_2 \rangle} [\mathtt{Nat}(zero), \mathtt{List}(cons(x_2))] \xrightarrow{\langle x_1, nil \rangle} [\,] \qquad (10.2)$$

is a refutation subtree of $[\mathtt{List}(cons(x_1, x_2))]$ with answer $\langle x_1, nil \rangle \circ \langle zero, x_2 \rangle = \langle zero, nil \rangle$. Observe that $[\mathtt{Nat}(x_1)]$ has a refutation subtree with the same answer: indeed for any substitution $\theta\colon 2 \to m$,

$$[\mathtt{Nat}(x_1)] \xrightarrow{\langle zero, x_2 \rangle} [\,] \xrightarrow{\theta} [\,] \qquad (10.3)$$

is a refutation subtree with answer $\theta \circ \langle zero, x_2 \rangle$. This is because rule $(l3)$ yields $[\,] \xrightarrow{\theta} [\,]$, which is graphically represented in our figures by an unlabeled edge.

It is interesting to note that concatenating (10.3) (where $\theta = \langle x_1, nil \rangle$) with (10.2) via $\overline{::}$ yields a refutation subtree for $[\mathtt{Nat}(x_1), \mathtt{List}(cons(x_1, x_2))]$ with the same answer: this is

$$[\mathtt{Nat}(x_1), \mathtt{List}(cons(x_1, x_2))] \xrightarrow{\langle zero, x_2 \rangle} [\mathtt{Nat}(zero), \mathtt{List}(cons(x_2))] \xrightarrow{\langle x_1, nil \rangle} [\,] \qquad (10.4)$$

depicted at the center of Figure 3. More generally, if two trees T_1 and T_2 have refutation subtrees with the same sequence of substitutions $\theta_k \circ \ldots \theta_2 \circ \theta_1$, then also $T_1 \overline{::} T_2$ has a refutation subtree with the same sequence.

For an example involving r_{sat}, consider part of the saturated ∧∨-tree for $\mathtt{List}(cons(x_1, x_2))$ depicted below (Example 4.9 discusses a different part of the same tree). This is mapped by r_{sat} into the saturated ∨-tree of $[\mathtt{List}(cons(x_1, x_2))]$ shown on the right of Figure 2.
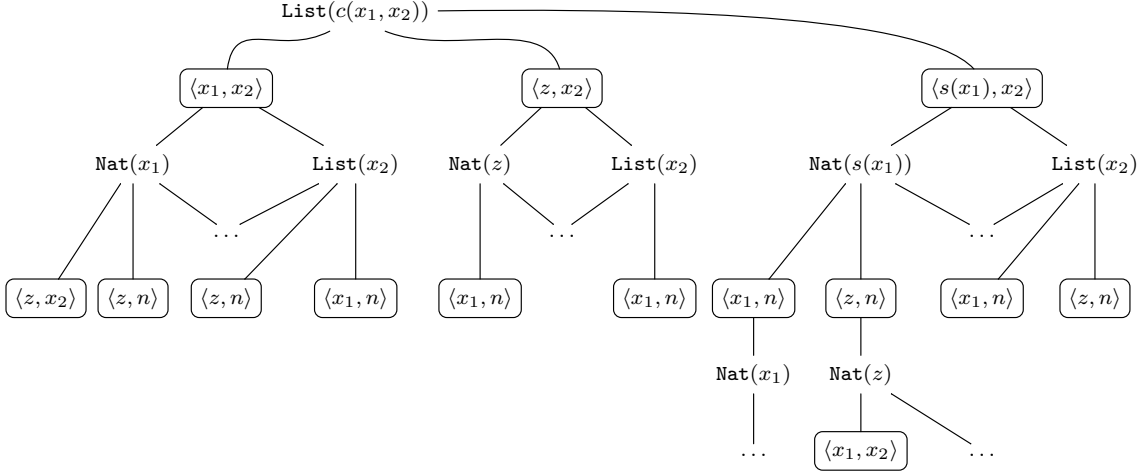
The representation map r_{sat} behaves similarly to the one given for the ground case (*cf.* Example 8.7), the main difference being that, in saturated $\wedge\vee$-trees, $\vee$-nodes are now labeled with substitutions: the effect of r_{sat} is to move such substitutions to the edges of the target saturated $\vee$-trees. For instance, the label $\langle x_1, x_2 \rangle$ on the $\vee$-node (on the left above) is moved in Figure 2 to the edge connecting the root $[\texttt{List}(cons(x_1, x_2))]$ with the node labeled with $[\texttt{Nat}(x_1), \texttt{List}(x_2)]$. Observe that this node has one $\langle zero, nil \rangle$-child and no children associated with $\langle zero, x_2 \rangle$ or $\langle x_1, nil \rangle$: instead, those two substitutions label one child of $\texttt{Nat}(x_1)$ and one of $\texttt{List}(x_2)$, respectively (on the left above). Intuitively, the children of $[\texttt{Nat}(x_1), \texttt{List}(x_2)]$ are given by considering only the children of $\texttt{Nat}(x_1)$ and those of $\texttt{List}(x_2)$ labeled with the same substitution.

It is worth to observe that, for every synched refutation subtree T' (Definition 6.1) of a saturated $\wedge\vee$-tree T there is a refutation subtree in $r_{sat}(T)$ with the same answer. The effect on T' of applying r_{sat} to T can be described by the following procedure: for every depth in T', (a) all the $\wedge$-nodes are grouped into a single node whose label lists all the labels of these $\wedge$-nodes; (b) the $\vee$-nodes become an edge whose label is the common substitution labeling all these $\vee$-nodes. For an example, we depict below a synched derivation subtree on the left and the corresponding subtree on the right. (In the saturated $\wedge\vee$-tree above there are other three synched refutation subtrees: the reader can find the corresponding refutation subtrees in the $\vee$-tree on the right of Figure 2.)
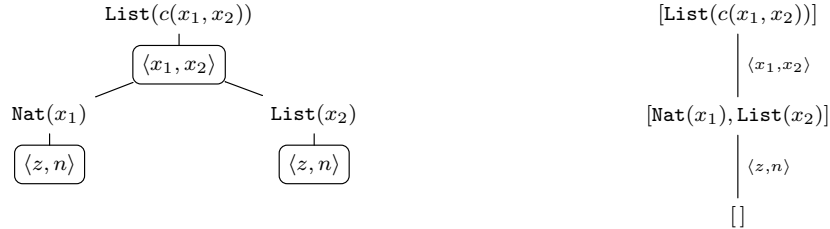
Such transformation is neither injective nor surjective. Against surjectivity, consider the refutation subtree (10.3): it does not correspond to any synched derivation subtree because it contains two occurrences of $[\,]$, while the above procedure always transforms synched derivation subtrees into refutation subtrees with exactly one occurrence of $[\,]$. Nevertheless, using the construction of Proposition 10.6 one can show that for every refutation subtree in $r_{sat}(T)$ there exists a synched refutation subtree in T with the same answer. For instance,

(10.3) corresponds to the synched refutation subtree with root $\mathtt{Nat}(x_1)$ and the only other node a child of $\mathtt{Nat}(x_1)$ labeled with $\theta \circ \langle zero, x_2 \rangle$.

Generalizing the observations of Example 10.7 above, we have that a saturated $\wedge\vee$-tree T has a synched derivation subtree with answer θ if and only if $r_{sat}(T)$ has a refutation subtree with answer θ. In case $T = [\![A]\!]_{p^\sharp}$ for some atom $A \in At(n)$, then $r_{sat}(T) = [\![[A]\!]]_{p_\delta^\sharp}$ by Proposition 10.4. This yields the following statement.

Proposition 10.8. *Fix an atom $A \in At(n)$ and a program $\mathbb{P}$. The following are equivalent.*

(I) The saturated $\vee$-tree for $[A] \in \check{\mathcal{L}} At(n)$ in $\mathbb{P}$ has a refutation subtree with answer θ.

(II) The saturated $\wedge\vee$-tree for A in $\mathbb{P}$ has a synched refutation subtree with answer θ.

Corollary 10.9 (Soundness and Completeness I). *Let $\mathbb{P}$ be a logic program and $A \in At(n)$ an atom. The following statement is equivalent to any of the three of Theorem 6.2.*

(4) The saturated $\vee$-tree for $[A] \in \check{\mathcal{L}} At(n)$ in $\mathbb{P}$ has a refutation subtree with answer θ.

Proof. It suffices to prove the equivalence between (4) and statement (1) of Theorem 6.2, which is given by Proposition 10.8. $\qquad\square$

In fact, since both bialgebraic semantics and SLD-resolution are defined on arbitrary goals, we can state the following stronger result.

Theorem 10.10 (Soundness and Completeness II). *Let $\mathbb{P}$ be a logic program and $G = [A_1, \ldots, A_k] \in \check{\mathcal{L}} At(n)$ be a goal. The following are equivalent.*

(1) The saturated $\vee$-tree for G in $\mathbb{P}$ has a refutation subtree with answer θ.

(2) There is an SLD-refutation for G in $\mathbb{P}$ with correct answer θ.

Proof. Fix a program $\mathbb{P}$ and the goal $G = [A_1, \ldots, A_k]$. The statement is given by the following reasoning.

G has an SLD-refutation

with correct answer θ	$\Leftrightarrow$	each $[A_i]$ has an SLD-refutation with correct answer θ
(Corollary 10.9)	$\Leftrightarrow$	each $[\![[A_i]\!]]_{p_\delta^\sharp}$ has a refutation subtree with answer θ
	$\Leftrightarrow$	$[\![[A_1]\!]]_{p_\delta^\sharp} \mathbin{\ddot{::}} \ldots \mathbin{\ddot{::}} [\![[A_k]\!]]_{p_\delta^\sharp}$ has a refutation subtree with answer θ
(Theorem 9.9)	$\Leftrightarrow$	$[\![G]\!]_{p_\delta^\sharp}$ has a refutation subtree with answer θ.

The first equivalence is a basic fact stemming from the definition of SLD-resolution. The third equivalence comes from the observation that, analogously to the ground case, $\mathbin{\ddot{::}}$ preserves and reflects the property of yielding a refutation. To see this, suppose that $[\![[A_k]\!]]_{p_\delta^\sharp}, \ldots, [\![[A_k]\!]]_{p_\delta^\sharp}$ all have refutation subtrees with answer $\theta \colon n \to m$. By Proposition 10.6, for each i we can pick a refutation subtree T_i of $[\![[A_i]\!]]_{p_\delta^\sharp}$ with answer θ, such that θ is the substitution labeling the edge connecting depth 0 and 1 and all the other edges are labeled with id_m. Therefore, at each depth, $T_1, \ldots, T_k$ all have the same substitution labeling the corresponding edge. This means that the operation $\mathbin{\ddot{::}}$ applied to $[\![[A_1]\!]]_{p_\delta^\sharp}, \ldots, [\![[A_k]\!]]_{p_\delta^\sharp}$ has the effect of "gluing together" $T_1, \ldots, T_k$ into a refutation subtree T of $[\![[A_1]\!]]_{p_\delta^\sharp} \mathbin{\ddot{::}} \ldots \mathbin{\ddot{::}} [\![[A_k]\!]]_{p_\delta^\sharp}$ with answer θ, just as in Example 10.7, where (10.3) and (10.2) were glued to form (10.4). Starting instead with a refutation subtree in $[\![[A_1]\!]]_{p_\delta^\sharp} \mathbin{\ddot{::}} \ldots \mathbin{\ddot{::}} [\![[A_k]\!]]_{p_\delta^\sharp}$, one clearly has a decomposition in the converse direction. $\qquad\square$

11. Conclusions

The first part of this work proposed a coalgebraic semantics for logic programming, extending the framework introduced in [30] for the case of ground logic programs. Our approach has been formulated in terms of coalgebrae on presheaves, whose nice categorical properties made harmless to reuse the very same constructions as in the ground case. A critical point of this generalization was to achieve compositionality with respect to substitutions, which we obtained by employing *saturation* techniques. We emphasized how these can be explained in terms of substitution mechanisms: while the operational semantics p proposed in [32] is associated with term-matching, its saturation $p^\sharp$ corresponds to unification. The map $p^\sharp$ gave rise to the notion of *saturated $\wedge\vee$-tree*, as the model of computation represented in our semantics. We observed that coinductive trees, introduced in [32], can be seen as a desaturated version of saturated $\wedge\vee$-trees, and we compared the two notions with a translation. Eventually, we tailored a notion of subtree (of a saturated $\wedge\vee$-tree), called *synched derivation subtree*, representing a sound derivation of a goal in a program. This led to a result of soundness and completeness of our semantics with respect to SLD-resolution.

In the second part of the paper, we extended our framework to model the saturated semantics of goals with bialgebrae on presheaves. The main feature of this approach was yet another form of compositionality: the semantics of a goal G can be equivalently expressed as the "pasting" of the semantics of the single atoms composing G. This property arose naturally via universal categorical constructions based on monads and distributive laws. The corresponding operational description was given the name of *saturated $\vee$-trees*. The synchronisation of different branches of a derivation subtree, which was imposed on saturated $\wedge\vee$-trees, is now given by construction: in saturated $\vee$-trees the parallel resolution of each atom in the goal always proceeds with the same substitution. On the base of these observations, we extended the soundness and completeness result for saturated semantics to the SLD-resolution of arbitrary (and not just atomic) goals.

Saturated $\vee$-trees carry more information than traditional denotational models like Herbrand or C-models [20]. The latter can be obtained by saturated $\vee$-trees as follows: a substitution θ is an answer of the saturated $\vee$-tree of a goal G if and only if $G\theta$ belongs to the minimal C-model. For future work, we would like to find the right categorical machinery to transform saturated $\vee$-trees into C-models. The approach should be close to the one used in [7] for the semantics of non-deterministic automata with ϵ-transitions: first, the branching structure of $\vee$-trees is flattened into sets of sequences of substitutions (similarly to passing from bisimilarity to trace equivalence); second, the substitutions in a sequence are composed to form a single substitution (similarly to composing a sequence of words to form a single word).

Moreover, we find of interest to investigate *infinite* computations and the semantics of coinductive logic programming [25]. These have been fruitfully explored within the approach based on coinductive trees [31]. We expect our analysis of the notion of synchronisation for derivation subtrees to bring further insights on the question.

Another line of research concerns a deeper understanding of saturation. A drawback of saturated semantics is that one has to take into account *all* the arrows in the index category, which are usually infinitely many. This problem has been tackled in [8] for powerset-like functors, by employing *normalized coalgebrae*: one considers only a *minimal* set of arrows, as an "optimal" representation of all the arrows in the index category. It is not immediate to see how this approach can be generalized to arbitrary functors, as it would be in the

formulation of saturation in terms of right Kan extension. We believe that the case of logic programming represents an ideal benchmark to investigate this question. Indeed, minimal arrows exactly correspond to most general unifiers. If we understand how the minimal arrow approach works in this setting, we could take aim at a direct categorical modeling of SLD-resolution and its tree representation.

References

[1] L. Aceto, W. Fokkink, and C. Verhoef. Structural operational semantics. In *Handbook of Process Algebra*. Elsevier, 2001.

[2] J. Adámek and V. Koubek. On the greatest fixed point of a set functor. *Theor. Comput. Sci.*, 150:57–75, 1995.

[3] J. Adámek and J. Rosický. *Locally Presentable and Accessible Categories*. Cambridge University Press, 1994.

[4] G. Amato, J. Lipton, and R. McGrail. On the algebraic structure of declarative programming languages. *Theor. Comput. Sci.*, 410(46):4626–4671, 2009.

[5] K. R. Apt and M. H. van Emden. Contributions to the theory of logic programming. *J. ACM*, 29(3):841–862, July 1982.

[6] F. Bonchi, M. G. Buscemi, V. Ciancia, and F. Gadducci. A presheaf environment for the explicit fusion calculus. *J. Autom. Reason.*, 49(2):161–183, 2012.

[7] F. Bonchi, S. Milius, A. Silva, and F. Zanasi. How to kill epsilons with a dagger - A coalgebraic take on systems with algebraic label structure. In *Coalgebraic Methods in Computer Science - 12th IFIP WG 1.3 International Workshop, CMCS 2014, Colocated with ETAPS 2014, Grenoble, France, April 5-6, 2014, Revised Selected Papers*, pages 53–74, 2014.

[8] F. Bonchi and U. Montanari. Coalgebraic symbolic semantics. In *CALCO*, volume 5728 of *LNCS*, pages 173–190. Springer, 2009.

[9] F. Bonchi and U. Montanari. Reactive systems, (semi-)saturated semantics and coalgebras on presheaves. *Theor. Comput. Sci.*, 410(41):4044–4066, 2009.

[10] F. Bonchi and F. Zanasi. Saturated semantics for coalgebraic logic programming. In *CALCO*, pages 80–94, 2013.

[11] R. Bruni, H. C. Melgratti, U. Montanari, and P. Sobocinski. Connector algebras for C/E and P/T nets' interactions. *Logical Methods in Computer Science*, 9(3), 2013.

[12] R. Bruni, U. Montanari, and F. Rossi. An interactive semantics of logic programming. *Theory and Practice of Logic Programming*, 1(6):647–690, 2001.

[13] L. Cardelli and A. D. Gordon. Mobile ambients. In M. Nivat, editor, *FoSSaCS*, volume 1378 of *Lecture Notes in Computer Science*, pages 140–155. Springer, 1998.

[14] E. Cheng. Iterated distributive laws. *Math. Proc. Camb. Philos. Soc.*, 150(3):459–487, 2011.

[15] K. Clark. *Predicate Logic as a Computational Formalism*. Research monograph / Department of Computing, Imperial College of Science and Technology, University of London. University of London, 1980.

[16] A. Corradini, M. Große-Rhode, and R. Heckel. A coalgebraic presentation of structured transition systems. *Theor. Comput. Sci.*, 260(1-2):27–55, 2001.

[17] A. Corradini, R. Heckel, and U. Montanari. From SOS specifications to structured coalgebras: How to make bisimulation a congruence. *ENTCS*, 19(0):118 – 141, 1999.

[18] A. Corradini and U. Montanari. An algebraic semantics for structured transition systems and its application to logic programs. *Theor. Comput. Sci.*, 103(1):51 – 106, 1992.

[19] C. Dwork, P. C. Kanellakis, and J. C. Mitchell. On the sequential nature of unification. *The Journal of Logic Programming*, 1(1):35 – 50, 1984.

[20] M. Falaschi, G. Levi, C. Palamidessi, and M. Martelli. Declarative modeling of the operational behavior of logic languages. *Theoretical Computer Science*, 69(3):289 – 318, 1989.

[21] M. P. Fiore, E. Moggi, and D. Sangiorgi. A fully abstract model for the π-calculus. *Inf. Comput.*, 179(1):76–117, 2002.

[22] M. P. Fiore and S. Staton. A congruence rule format for name-passing process calculi from mathematical structural operational semantics. In *LICS*, pages 49–58. IEEE, 2006.

[23] M. P. Fiore and D. Turi. Semantics of name and value passing. In *LICS*, pages 93–104. IEEE, 2001.

[24] J. A. Goguen. What is unification? - a categorical view of substitution, equation and solution. In *Resolution of Equations in Algebraic Structures, Volume 1: Algebraic Techniques*, pages 217–261. Academic, 1989.

[25] G. Gupta, A. Bansal, R. Min, L. Simon, and A. Mallya. Coinductive logic programming and its applications. In *ICLP*, pages 27–44, 2007.

[26] G. Gupta and V. S. Costa. Optimal implementation of and-or parallel prolog. *Future Generation Computer Systems*, 10(1):71–92, 1994.

[27] B. Jacobs, A. Silva, and A. Sokolova. Trace semantics via determinization. In *CMCS*, pages 109–129, 2012.

[28] Y. Kinoshita and A. J. Power. A fibrational semantics for logic programs. In *Extensions of Logic Programming*, pages 177–191. Springer, 1996.

[29] B. Klin. Bialgebras for structural operational semantics: An introduction. *Theor. Comput. Sci.*, 412(38):5043–5069, 2011.

[30] E. Komendantskaya, G. McCusker, and J. Power. Coalgebraic semantics for parallel derivation strategies in logic programming. In *AMAST*, volume 6486 of *LNCS*, pages 111–127. Springer, 2011.

[31] E. Komendantskaya and J. Power. Coalgebraic derivations in logic programming. In *CSL*, volume 12 of *LIPIcs*, pages 352–366. Schloss Dagstuhl, 2011.

[32] E. Komendantskaya and J. Power. Coalgebraic semantics for derivations in logic programming. In *CALCO*, volume 6859 of *LNCS*, pages 268–282. Springer, 2011.

[33] E. Komendantskaya, J. Power, and M. Schmidt. Coalgebraic logic programming: from semantics to implementation. *Journal of Logic and Computation*, 2014.

[34] J. J. Leifer and R. Milner. Deriving bisimulation congruences for reactive systems. In *CONCUR*, volume 1877, pages 243–258, 2000.

[35] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2nd edition, 1993.

[36] S. Mac Lane. *Categories for the Working Mathematician*. Graduate Texts in Mathematics. Springer, 2nd edition, Sept. 1998.

[37] S. MacLane and I. Moerdijk. *Sheaves in Geometry and Logic: A First Introduction to Topos Theory*. Springer, corrected edition, May 1992.

[38] E. Manes and P. Mulry. Monad compositions I: general constructions and recursive distributive laws. *Theory and Applications of Categories*, 18(7):172–208, 2007.

[39] M. Miculan. A categorical model of the fusion calculus. *ENTCS*, 218:275–293, 2008.

[40] M. Miculan and K. Yemane. A unifying model of variables and names. In *FOSSACS*, volume 3441 of *LNCS*, pages 170–186. Springer, 2005.

[41] U. Montanari and M. Sammartino. A network-conscious pi-calculus and its coalgebraic semantics. *Submitted to TCS (Festschrift for Glynn Winskel)*.

[42] U. Montanari and V. Sassone. Dynamic congruence vs. progressing bisimulation for CCS. *FI*, 16(1):171–199, 1992.

[43] P. S. Mulry. Lifting theorems for Kleisli categories. In *MFPS*, pages 304–319, 1993.

[44] A. W. Roscoe, C. A. Hoare, and R. Bird. *The theory and practice of concurrency*, volume 169. Prentice Hall Englewood Cliffs, 1998.

[45] J. J. M. M. Rutten. Universal coalgebra: a theory of systems. *Theor. Comput. Sci.*, 249(1):3–80, 2000.

[46] D. Sangiorgi. A theory of bisimulation for the pi-calculus. *Acta Inf.*, 33(1):69–97, 1996.

[47] A. Silva, F. Bonchi, M. M. Bonsangue, and J. J. M. M. Rutten. Generalizing the powerset construction, coalgebraically. In *FSTTCS*, pages 272–283, 2010.

[48] S. Staton. Relating coalgebraic notions of bisimulation. In *CALCO*, volume 5728 of *LNCS*, pages 191–205. Springer, 2009.

[49] M. Tanaka. *Pseudo-distributive Laws and a Unified Framework for Variable Binding*. PhD thesis, University of Edinburgh, 2005.

[50] D. Turi and G. D. Plotkin. Towards a mathematical operational semantics. In *LICS*, pages 280–291. IEEE Computer Society, 1997.

[51] M. H. Van Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. *J. ACM*, 23(4):733–742, Oct. 1976.

[52] J. Worrell. Terminal sequences for accessible endofunctors. *ENTCS*, 19:24–38, 1999.

APPENDIX A. PROOFS

In this appendix we collect the proofs of some of the statements in the main text, whose details are not of direct relevance for our exposition.

Proposition 6.3. Let $\mathbb{P}$ be a logic program and $A \in At(n)$ an atom. If $[\![A]\!]_{p^\sharp}$ has a synched refutation subtree with answer $\theta \colon n \to m$, then $[\![A]\!]_{p^\sharp}\overline{\theta}$ has a synched refutation subtree whose $\vee$-nodes are all labeled with id_m.

Proof. First, we fix the two following properties, holding for all the $\wedge$-nodes of $[\![A]\!]_{p^\sharp}$.

- (†) Let θ, θ' be two arrows in $\mathbf{L}_{\boldsymbol{\Sigma}}^{op}$ such that $\theta' \circ \theta$ is defined. If an $\wedge$-node s has a child t such that (a) the label of t is θ and (b) t has children labeled with $B_1, \ldots, B_n$, then s has also another child t' such that (a) the label of t' is $\theta' \circ \theta$ and (b) t' has children labeled with $B_1\theta', \ldots, B_n\theta'$.
- (‡) Let $\theta, \theta', \sigma, \sigma'$ be four arrows in $\mathbf{L}_{\boldsymbol{\Sigma}}^{op}$ such that $\sigma \circ \theta = \sigma' \circ \theta'$. If an $\wedge$-node labeled with A' has a child t such that (a) the label of t is θ and (b) t has children labeled with $B_1, \ldots, B_n$, then each node labeled with $A'\theta'$ has a child t' such that (a) the label of t' is σ' and (b) t' has children labeled with $B_1\sigma, \ldots, B_n\sigma$.

Assume that $[\![A]\!]_{p^\sharp}$ has a synched refutation subtree T whose $\vee$-nodes are labeled with θ_1, θ_3, $\ldots$, θ_{2k+1} (where θ_i is the substitution labeling the $\vee$-nodes of depth i). We prove that $[\![A]\!]_{p^\sharp}$ has another synched refutation subtree T' whose first $\vee$-node is labeled with $\theta = \theta_{2k+1} \circ \theta_{2k-1} \circ \ldots \circ \theta_1$ and all the other $\vee$-nodes are labeled with identities.

By assumption, the root r has a child (in T) that is labeled with θ_1. Assume that its children are labeled with $B_1^2 \ldots B_{n_2}^2$. By (†), r has another child t' (in $[\![A]\!]_{p^\sharp}$), that (a) is labeled with θ and (b) has children labeled with $B_1^2\sigma_3 \ldots B_n^2\sigma_3$ where $\sigma_3 = \theta_{k+1}\circ\theta_{k-1}\circ\ldots\circ\theta_3$. These children form depth 2 of T' (the root r and t form, respectively, depth 0 and 1).

We now build the other depths. For an even $i \le 2k$, let σ_{i+1} denote $\theta_{2k+1}\circ\theta_{2k-1}\circ\cdots\circ\theta_{i+1}$ and let $B_1^i, \ldots, B_{n_i}^i$ be the labels of the $\wedge$-nodes of T at depth i. The depth i of T' is given by $\wedge$-nodes labeled with $B_1^i\sigma_{i+1}, \ldots, B_{n_i}^i\sigma_{i+1}$; the depth $i+1$ by $\vee$-nodes all labeled with id_m. It is easy to see that T' is a subtree of $[\![A]\!]_{p^\sharp}$: by assumption the nodes labeled with $B_1^i, \ldots, B_{n_i}^i$ have children in T all labeled with θ_{i+1}; since $\sigma_{i+3} \circ \theta_{i+1} = id_m \circ \sigma_{i+1}$, by property (‡), the nodes labeled with $B_1^i\sigma_{i+1}, \ldots, B_{n_i}^i\sigma_{i+1}$ have children (in $[\![A]\!]_{p^\sharp}$) that (a) are labeled with id_m and (b) have children with labels $B_1^{i+2}\sigma_{i+3}, \ldots, B_{n_{i+2}}^{i+2}\sigma_{i+3}$.

Once we have built T', we can easily conclude. Recall that t' (the first $\vee$-node of T') is labeled with θ. Following the construction at the end of Section 4, the root of $[\![A]\!]_{p^\sharp}\overline{\theta}$ has a child that is labeled with id_m and that has the same children as t'. Therefore $[\![A]\!]_{p^\sharp}\overline{\theta}$ has a synched refutation subtree with answer id_m. $\square$

Next we provide a proof of the following statement.

Proposition 9.3. Let $\mathbf{C}$ and $\mathbf{D}$ be categories with an adjunction $\mathcal{U} \dashv \mathcal{K}$ for $\mathcal{U} \colon \mathbf{C} \to \mathbf{D}$ and $\mathcal{K} \colon \mathbf{D} \to \mathbf{C}$. Let $\breve{\mathcal{T}} \colon \mathbf{C} \to \mathbf{C}$ and $\widehat{\mathcal{T}} \colon \mathbf{D} \to \mathbf{D}$ be two monads such that $\mathcal{U}\breve{\mathcal{T}} = \widehat{\mathcal{T}}\mathcal{U}$. Then, there is a distributive law of monads $\lambda \colon \breve{\mathcal{T}}(\mathcal{K}\mathcal{U}) \Rightarrow (\mathcal{K}\mathcal{U})\breve{\mathcal{T}}$ defined for all $X \in \mathbf{C}$ by

$$(\widehat{\mathcal{T}}(id_{\mathcal{K}\mathcal{U}X})^\flat)^\sharp$$

where $(\cdot)_{X,Z}^\flat \colon \mathbf{C}[X, \mathcal{K}Z] \to \mathbf{D}[\mathcal{U}X, Z]$ and $(\cdot)_{X,Z}^\sharp \colon \mathbf{D}[\mathcal{U}X, Z] \to \mathbf{C}[X, \mathcal{K}Z]$ are the components of the canonical bijection given by the adjunction $\mathcal{U} \dashv \mathcal{K}$.

For this purpose, we first need to state some auxiliary preliminaries on monads and distributive laws. Given a monad $(\mathcal{M}, \eta^{\mathcal{M}}, \mu^{\mathcal{M}})$ on a category $\mathbf{C}$, we use $\mathcal{K}\ell(\mathcal{M})$ to denote its Kleisli category and $\mathcal{J} \colon \mathbf{C} \to \mathcal{K}\ell(\mathcal{M})$ to denote the canonical functor acting as identity on objects and mapping $f \in \mathbf{C}[X, Y]$ into $\eta_Y \circ f \in \mathcal{K}\ell(\mathcal{K}\mathcal{U})[X, Y]$. The *lifting* of a monad $(\mathcal{T}, \eta, \mu)$ in $\mathbf{C}$ is a monad $(\mathcal{T}', \eta', \mu')$ in $\mathcal{K}\ell(\mathcal{M})$ such that $\mathcal{J}\mathcal{T} = \mathcal{T}'\mathcal{J}$ and η', μ' are given on $\mathcal{J}X \in \mathcal{K}\ell(\mathcal{M})$ (i.e. $X \in \mathbf{C}$) respectively as $\mathcal{J}(\eta_X)$ and $\mathcal{J}(\mu_X)$.

The following "folklore" result gives an alternative description of distributive laws in terms of liftings to Kleisli categories, see e.g. [49, §4], [43].

Proposition A.1. *Let $(\mathcal{M}, \eta^{\mathcal{M}}, \mu^{\mathcal{M}})$ be a monad on a category $\mathbf{C}$. For every monad $(\mathcal{T}, \eta^{\mathcal{T}}, \mu^{\mathcal{T}})$ on $\mathbf{C}$, there is a bijective correspondence between liftings of $(\mathcal{T}, \eta^{\mathcal{T}}, \mu^{\mathcal{T}})$ to $\mathcal{K}\ell(\mathcal{M})$ and distributive laws of the monad $\mathcal{T}$ over the monad $\mathcal{M}$.*

We are now ready to supply the proof of Proposition 9.3.

Proof of Proposition 9.3. By Proposition A.1, it suffices that we define a monad lifting $\breve{\mathcal{T}}$ to $\mathcal{K}\ell(\mathcal{K}\mathcal{U})$. For this purpose, we will use the canonical comparison functor $\mathcal{H} \colon \mathcal{K}\ell(\mathcal{K}\mathcal{U}) \to \mathbf{D}$ associated with $\mathcal{K}\ell(\mathcal{K}\mathcal{U})$ (see e.g. [36, §VI.5]), enjoying the following property:

$$\mathcal{H}\mathcal{J} = \mathcal{U}. \tag{A.1}$$

Given $X, Y \in |\mathcal{K}\ell(\mathcal{K}\mathcal{U})|$ and $f \in \mathcal{K}\ell(\mathcal{K}\mathcal{U})[X, Y]$, the functor $\mathcal{H}$ is defined as

$$
\begin{array}{rrcl}
\mathcal{H} \colon & \mathcal{K}\ell(\mathcal{K}\mathcal{U}) & \to & \mathbf{D} \\
& X & \mapsto & \mathcal{U}X \\
& X \xrightarrow{f} Y & \mapsto & \mathcal{U}X \xrightarrow{f^\flat} \mathcal{U}Y
\end{array}
$$

where $f^\flat$ is given by observing that f in $\mathcal{K}\ell(\mathcal{K}\mathcal{U})$ is a morphism $f \colon X \to \mathcal{K}\mathcal{U}Y$ in $\mathbf{C}$ and using the bijective correspondence $(\cdot)^\flat_{X,Z} \colon \mathbf{C}[X, \mathcal{K}Z] \to \mathbf{D}[\mathcal{U}X, Z]$.

We have now the ingredients to introduce the monad $\ddot{\mathcal{T}}$ on $\mathcal{K}\ell(\mathcal{K}\mathcal{U})$ that we will later show to be a lifting of $\breve{\mathcal{T}}$. On objects $\mathcal{J}X$ of $\mathcal{K}\ell(\mathcal{K}\mathcal{U})$, the functor $\ddot{\mathcal{T}} \colon \mathcal{K}\ell(\mathcal{K}\mathcal{U}) \to \mathcal{K}\ell(\mathcal{K}\mathcal{U})$ is defined as $\mathcal{J}\breve{\mathcal{T}}X$ (note that all objects in $\mathcal{K}\ell(\mathcal{K}\mathcal{U})$ are of the shape $\mathcal{J}X$ for some $X \in \mathbf{C}$). For an arrow $\mathcal{J}X \xrightarrow{f} \mathcal{J}Y$ in $\mathcal{K}\ell(\mathcal{K}\mathcal{U})$, (i.e., $X \xrightarrow{f} \mathcal{K}\mathcal{U}Y$ in $\mathbf{C}$), we take $\mathcal{U}X \xrightarrow{f^\flat} \mathcal{U}Y$ in $\mathbf{D}$ and apply $\widehat{\mathcal{T}}$ to obtain $\widehat{\mathcal{T}}\mathcal{U}X \xrightarrow{\widehat{\mathcal{T}}f^\flat} \widehat{\mathcal{T}}\mathcal{U}Y$ which, by assumption, is $\mathcal{U}\breve{\mathcal{T}}X \xrightarrow{\widehat{\mathcal{T}}f^\flat} \mathcal{U}\breve{\mathcal{T}}Y$. Using the bijective correspondence $(\cdot)^\sharp_{X,Z} \colon \mathbf{D}[UX, Z] \to \mathbf{C}[X, \mathcal{K}Z]$, we obtain $\breve{\mathcal{T}}X \xrightarrow{(\widehat{\mathcal{T}}f^\flat)^\sharp} \mathcal{K}\mathcal{U}\breve{\mathcal{T}}Y$ in $\mathbf{C}$, that is an arrow $\mathcal{J}\breve{\mathcal{T}}X \xrightarrow{(\widehat{\mathcal{T}}f^\flat)^\sharp} \mathcal{J}\breve{\mathcal{T}}Y$ in $\mathcal{K}\ell(\mathcal{K}\mathcal{U})$. We define $\ddot{\mathcal{T}}f$ as $(\widehat{\mathcal{T}}f^\flat)^\sharp$. The unit of the monad $\eta^{\ddot{\mathcal{T}}}_X$ is defined as $X = \mathcal{J}X \xrightarrow{\mathcal{J}\eta^{\breve{\mathcal{T}}}_X} \mathcal{J}\breve{\mathcal{T}}X = \ddot{\mathcal{T}}X$. The multiplication $\mu^{\ddot{\mathcal{T}}}_X$ as $\ddot{\mathcal{T}}\ddot{\mathcal{T}}X = \mathcal{J}\breve{\mathcal{T}}\breve{\mathcal{T}}X \xrightarrow{\mathcal{J}\mu^{\breve{\mathcal{T}}}_X} \mathcal{J}\breve{\mathcal{T}}X = \ddot{\mathcal{T}}X$. One can readily check that $(\ddot{\mathcal{T}}, \eta^{\ddot{\mathcal{T}}}, \mu^{\ddot{\mathcal{T}}})$ is a monad.

In order to verify that $(\ddot{\mathcal{T}}, \eta^{\ddot{\mathcal{T}}}, \mu^{\ddot{\mathcal{T}}})$ is a monad lifting of $(\breve{\mathcal{T}}, \eta^{\breve{\mathcal{T}}}, \mu^{\breve{\mathcal{T}}})$, it only remains to check that $\ddot{\mathcal{T}}\mathcal{J} = \mathcal{J}\breve{\mathcal{T}}$, since the unit and multiplication are simply defined by applying $\mathcal{J}$ to

$\eta^{\breve{\mathfrak{T}}}$ and $\mu^{\breve{\mathfrak{T}}}$. For objects, it follows from the definition of $\ddot{\mathfrak{T}}$. For arrows $f \in \mathbf{C}[X, Y]$,

$$\ddot{\mathfrak{T}}\mathfrak{J}X \overset{\ddot{\mathfrak{T}}\mathfrak{J}f}{\rightarrow} \ddot{\mathfrak{T}}\mathfrak{J}Y = \qquad \mathfrak{J}\breve{\mathfrak{T}}X \overset{(\widehat{\mathfrak{T}}(\mathfrak{J}f)^{\flat})^{\sharp}}{\rightarrow} \mathfrak{J}\breve{\mathfrak{T}}Y \qquad \text{(definition of } \ddot{\mathfrak{T}})$$

$$= \qquad \mathfrak{J}\breve{\mathfrak{T}}X \overset{(\widehat{\mathfrak{T}}\mathcal{H}\mathfrak{J}f)^{\sharp}}{\rightarrow} \mathfrak{J}\breve{\mathfrak{T}}Y \qquad \text{(definition of } \mathcal{H})$$

$$= \qquad \mathfrak{J}\breve{\mathfrak{T}}X \overset{(\widehat{\mathfrak{T}}\mathcal{U}f)^{\sharp}}{\rightarrow} \mathfrak{J}\breve{\mathfrak{T}}Y \qquad \text{(by A.1)}$$

$$= \qquad \mathfrak{J}\breve{\mathfrak{T}}X \overset{(\mathcal{U}\breve{\mathfrak{T}}f)^{\sharp}}{\rightarrow} \mathfrak{J}\breve{\mathfrak{T}}Y \qquad \text{(by assumption)}$$

$$= \qquad \mathfrak{J}\breve{\mathfrak{T}}X \overset{(\mathcal{H}\mathfrak{J}\breve{\mathfrak{T}}f)^{\sharp}}{\rightarrow} \mathfrak{J}\breve{\mathfrak{T}}Y \qquad \text{(by A.1)}$$

$$= \qquad \mathfrak{J}\breve{\mathfrak{T}}X \overset{((\mathfrak{J}\breve{\mathfrak{T}}f)^{\flat})^{\sharp}}{\rightarrow} \mathfrak{J}\breve{\mathfrak{T}}Y \qquad \text{(definition of } \mathcal{H})$$

$$= \qquad \mathfrak{J}\breve{\mathfrak{T}}X \overset{\mathfrak{J}\breve{\mathfrak{T}}f}{\rightarrow} \mathfrak{J}\breve{\mathfrak{T}}Y. \qquad ((\cdot)^{\sharp} \text{ and } (\cdot)^{\flat} \text{ form a bijection)}$$

By Proposition A.1, we thus obtain a distributive law of monads $\lambda \colon \breve{\mathfrak{T}}(\mathcal{KU}) \Rightarrow (\mathcal{KU})\breve{\mathfrak{T}}$. Following the correspondence in [49], this is effectively constructed for all $X \in \mathbf{C}$ as follows. Let $id_{\mathcal{KU}X}$ be the identity on $\mathbf{C}$ and $\iota_X \colon \mathfrak{J}\mathcal{KU}X \to \mathfrak{J}X$ be the corresponding arrow in $\mathcal{K\ell}(\mathcal{KU})$. Then $\ddot{\mathfrak{T}}(\iota_X) \colon \mathfrak{J}\breve{\mathfrak{T}}\mathcal{KU}X \to \mathfrak{J}\breve{\mathfrak{T}}X$ in $\mathcal{K\ell}(\mathcal{KU})$ corresponds to an arrow $\ddot{\mathfrak{T}}(\iota_X) \colon \breve{\mathfrak{T}}\mathcal{KU}X \to \mathcal{KU}\breve{\mathfrak{T}}X$ in $\mathbf{C}$, which is how λ is defined on X. Unfolding the definition of $\ddot{\mathfrak{T}}$, this means that $\lambda_X = (\widehat{\mathfrak{T}}(id_{\mathcal{KU}X})^{\flat})^{\sharp}$. $\qquad \square$

Proposition 10.6. Let $\mathbb{P}$ be a logic program and $l \in \mathcal{L}At(n)$ a list of atoms. If $[\![l]\!]_{p_\delta^{\sharp}}$ has a refutation subtree with answer $\theta \colon n \to m$, then it has also a refutation subtree $T = s_1, s_2, \ldots$ with the same answer where the edge connecting s_1 and s_2 is labeled with θ and all the other edges in T are labeled with id_m.

Proof. The proof follows closely the one of Proposition 6.3. First, observe that properties analogous to (†) and (‡) in the proof of Proposition 6.3 hold for $\vee$-trees, since both saturated $\wedge\vee$- and $\vee$-trees are based on unification. We express them using the rule presentation of $p_\delta^{\sharp}$:

$$l4 \; \frac{l_1 \overset{\sigma}{\rightarrow} l_2}{l_1 \overset{\sigma' \circ \sigma}{\rightarrow} l_2\sigma'} \qquad l5 \; \frac{l_1 \overset{\tau}{\rightarrow} l_2 \qquad \sigma \circ \tau = \sigma' \circ \tau'}{l_1\tau' \overset{\sigma'}{\rightarrow} l_2\sigma}$$

where $l\theta$ is the result of applying θ to each atom in l. Rule (l4) corresponds to (†) and (l5) to (‡).

Now suppose that $T' = t_1, t_2, \ldots, t_k$ is a refutation subtree of $[\![l]\!]_{p_\delta^{\sharp}}$, say with answer $\theta = \theta_k \circ \ldots \circ \theta_2 \circ \theta_1$ and where t_i labeled with a list l_i. We build T inductively as follows. Depth 0 in T is given by the root t_1 labeled with $l_1 = l$. By construction θ_1 labels the edge between t_1 and t_2 in T'. Thus, by rule (l4), in $[\![l]\!]_{p_\delta^{\sharp}}$ the root t_1 has also an edge θ targeting a child s_2 with label $\theta_k \circ \ldots \circ \theta_2 l_2$. We let s_2 be the node of depth 1 in T. Inductively, suppose that we built T up to depth $i < k$. We know that in T' the node t_i is connected to t_{i+1} by an edge labeled with θ_i. Also, by inductive hypothesis the node s_i in T is labeled with $\theta_k \circ \ldots \circ \theta_i l_i$. Since $(\theta_k \circ \ldots \circ \theta_{i+1}) \circ \theta_i = id_m \circ (\theta_k \circ \ldots \circ \theta_i)$, then by rule (l5) the node s_i is connected to a node s_{i+1} in $[\![l]\!]_{p_\delta^{\sharp}}$ labeled with $\theta_k \circ \ldots \circ \theta_{i+1} l_{i+1}$ via an edge labeled with id_m. We let s_{i+1} be the node of depth $i + 1$ in T. It is clear by construction that T is a refutation subtree of $[\![l]\!]_{p_\delta^{\sharp}}$ with the required properties. $\qquad \square$