Saturated Semantics for Coalgebraic Logic Programming

Filippo Bonchi and Fabio Zanasi

ENS Lyon, U. de Lyon, CNRS, INRIA, UCBL

Abstract. A series of recent papers introduces a coalgebraic semantics for logic programming, where the behavior of a goal is represented by a *parallel* model of computation called coinductive tree. This semantics fails to be compositional, in the sense that the coalgebra formalizing such behavior does not commute with the substitutions that may apply to a goal. We suggest that this is an instance of a more general phenomenon, occurring in the setting of interactive systems (in particular, nominal process calculi), when one tries to model their semantics with coalgebrae on presheaves. In those cases, compositionality can be obtained through *saturation.* We apply the same approach to logic programming: the resulting semantics is compositional and enjoys an elegant formulation in terms of coalgebrae on presheaves and their right Kan extensions.

1 Introduction

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 [17, 14, 37, 16, 15]. 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 [18, 33, 38] or name fusions [32, 5] 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 $\alpha(t\theta) \neq \alpha(t)\overline{\theta}$, where $\overline{\theta}$ 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 other words, bisimilarity is not a congruence w.r.t. name substitutions. In order to make bisimilarity a congruence, Sangiorgi introduced in [36] *open bisimilarity*, that is defined by considering the transitions of processes under *all* possible name substitutions θ.

The approach of *saturated semantics* [6, 8] 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 [5, 34], 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 $(-)^\sharp_{X,Y}\colon \mathbf{Set}^{|\mathbf{C}|}[\mathcal{U}(X), Y] \to \mathbf{Set}^{\mathbf{C}}[X, \mathcal{K}(Y)]$ 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 this paper, we show that the saturated approach can be fruitfully instantiated to *coalgebraic logic programming* [23, 25, 24, 26], which consists of a novel semantics for logic programming and a parallel resolution algorithm based on *coinductive trees*. These are a variant of and-or trees [21] 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 [13]).

There are two analogies with the π-calculus: (a) the state space is modeled by a presheaf on the index category $\mathbf{L}_{\mathbf{\Sigma}}^{op}$, that is the (opposite) *Lawvere Theory* associated with some signature Σ; (b) the operational semantics given in [25] fails to be a natural transformation in $\mathbf{Set}^{\mathbf{L}_{\mathbf{\Sigma}}^{op}}$: Example 2 provides a counter-example which is similar to the π-calculus term t discussed above. The authors of [25] 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 $\mathit{Lax}(\mathbf{L}_{\mathbf{\Sigma}}^{op}, \mathbf{Poset})$ of locally ordered functors $\mathcal{F}\colon \mathbf{L}_{\mathbf{\Sigma}}^{op} \to \mathbf{Poset}$ and lax natural transformations between them. They show the existence of a cofree comonad that induces a morphism $[\![-]\!]_p$ mapping atoms (i.e., atomic formulae) to coinductive trees. Since $[\![-]\!]_p$ 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,

$$[\![A\theta]\!]_p \neq [\![A]\!]_p\overline{\theta}$$

where $[\![A\theta]\!]_p$ is the coinductive tree associated with $A\theta$ and $[\![A]\!]_p\overline{\theta}$ denotes the result of applying θ to each atom occurring in the tree $[\![A]\!]_p$.

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 $(-)^\sharp$ 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 morphism in $\mathbf{Set}^{\mathbf{L}_{\mathbf{\Sigma}}^{op}}$) 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 $[\![-]\!]_{p^\sharp}$ can be obtained by very standard categorical tools, such as terminal sequences [1]. This is possible because, as **Set**, both $\mathbf{Set}^{\mathbf{L}_{\mathbf{\Sigma}}^{op}}$ and $\mathbf{Set}^{|\mathbf{L}_{\mathbf{\Sigma}}^{op}|}$ are (co)complete categories, whereas in the lax approach, $Lax(\mathbf{L}_{\mathbf{\Sigma}}^{op}, \mathbf{Poset})$ not being (co)complete, more indirect and more sophisticated categorical constructions are needed [24, Sec. 4]. By naturality of $p^\sharp$, the semantics given by $[\![-]\!]_{p^\sharp}$ turns out to be compositional, as in the desiderata. Analogously to $[\![-]\!]_{p}$, also $[\![-]\!]_{p^\sharp}$ maps atoms to tree structures, which we call *saturated trees.* They generalize coinductive trees, in the sense that the latter can be seen as a "desaturation" of saturated 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 [25, 24].

Analogously to what is done in [24], we propose a notion of *refutation subtree* of a given saturated tree, intuitively corresponding to an SLD-refutation of an atomic goal in a program. 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.

Related works. Apart from [23, 25, 24], there exist other categorical perspectives on (extensions of) logic programming, such as [12, 22, 31, 3]. Amongst these, the most relevant for us is [8] since it exploits a form of saturation: states representing formulae are both instantiated by substitution and contextualized by other formulae in "and". Beyond logic programming, the idea of exploiting saturation to achieve compositionality is even older than [36] (see e.g. [35]). As far as we know, [11] is the first work where saturation is explored in terms of coalgebrae. It is interesting to note that, in [10], a subset 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 [32, 38] 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. [4]).

Synopsis. Section 2 introduces the background on logic programming and its coalgebraic semantics. In Section 3 we propose saturated semantics as an alternative to laxness and we show that it is compositional. Section 4 builds a bridge between the different approaches. We define a translation from saturated to coinductive trees. In Section 5 the results of the previous two sections are used to deduce soundness and completeness of saturated semantics with respect to SLD-resolution.

2 Coalgebraic Logic Programming

In this section we recall the framework of coalgebraic logic programming, as introduced in [23, 25, 24]. For this purpose, we first fix some terminology and notation, mainly concerning category theory and logic programming.

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, $o \in |\mathbf{C}|$ indicates that o is an object of $\mathbf{C}$ and $\mathbf{C}[o_1, o_2]$ the set of arrows from o_1 to o_2. 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 between them. Given a functor $\mathcal{B}\colon \mathbf{C} \to \mathbf{C}$, a $\mathcal{B}$-*coalgebra* on $o \in |\mathbf{C}|$ is an arrow $p\colon o \to \mathcal{B}(o)$.

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 [19, 9]. To this aim, let the (opposite) *Lawvere Theory* of Σ be a category $\mathbf{L}_{\mathbf{\Sigma}}^{op}$ where objects are natural numbers, with $n \in |\mathbf{L}_{\mathbf{\Sigma}}^{op}|$ intuitively representing variables $x_1, x_2, \dots, x_n$ from Var. For any two $n, m \in |\mathbf{L}_{\mathbf{\Sigma}}^{op}|$, the set $\mathbf{L}_{\mathbf{\Sigma}}^{op}[n, m]$ 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 $n \in |\mathbf{L}_{\mathbf{\Sigma}}^{op}|$, 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 $\mathbf{L}_{\mathbf{\Sigma}}^{op}$ 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 $\mathbf{L}_{\mathbf{\Sigma}}^{op}$, 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 $\{A_1, \dots, A_m\}\theta$ is a shorthand for $\{A_1\theta, \dots, A_m\theta\}$. 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 $A_1 = P(t_1, \dots, t_n)$, $A_2 = P(t'_1, \dots, t'_n)$ and a unifier $\langle \sigma, \tau \rangle$ of $\langle t_1, \dots, t_n \rangle$ and $\langle t'_1, \dots, t'_n \rangle$ 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 $A_1 = A_2\tau$.

A *logic program* $\mathbb{P}$ consists of a finite set of *clauses* C written as $H \leftarrow B_1, \dots, B_k$. The components H and $B_1, \dots, B_k$ are atoms, where H is called the *head* of C and $B_1, \dots, B_k$ form the *body* of C. One can think of $H \leftarrow B_1, \dots, B_k$ as representing the first-order formula $(B_1 \wedge \dots \wedge B_k) \to H$. 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 [27, 28], checking whether a finite set of atoms (called a *goal*) is *refutable* in $\mathbb{P}$ and giving a substitution called *computed answer* as output: we refer to Appendix A for more details. Relevant for our exposition are *and-or trees* [21], 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.

Definition 1. *Given a logic program $\mathbb{P}$ and an atom A, the* (parallel) and-or tree *for A in $\mathbb{P}$ is the possibly infinite tree T satisfying the following properties:*

1. *Each node in T is either an and-node or an or-node.*
2. *Each and-node is labeled with one atom and its children are or-nodes.*
3. *The root of T is an and-node labeled with A.*
4. *Each or-node is labeled with $\bullet$ and its children are and-nodes.*
5. *For every and-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.

2.1 The Ground Case

We recall the coalgebraic semantics of ground logic programs introduced in [23]. 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 and-or tree of A in $\mathbb{P}$ up to depth 2, according to Definition 1: 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 [1, 39].

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

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

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

By [39] it follows that the sequence given above converges to a limit X_γ such that $X_\gamma \cong X_{\gamma+1}$. 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 $\pi_2 \circ \delta_\gamma^{-1} \colon X_\gamma \to \mathcal{P}_f\mathcal{P}_f(X_\gamma)$ the cofree $\mathcal{P}_f\mathcal{P}_f$-coalgebra *on At. This induces the* cofree comonad $\mathcal{C}(\mathcal{P}_f\mathcal{P}_f) \colon$ **Set** $\to$ **Set** *on $\mathcal{P}_f\mathcal{P}_f$ as a functor mapping At to X_γ.*

As the elements of the cofree comonad on $\mathcal{P}_f$ are standardly presented as finitely branching trees [39], those for $\mathcal{P}_f\mathcal{P}_f$ can be seen as finitely branching trees with two sorts of nodes occurring at alternating depth. We now define a $\mathcal{C}(\mathcal{P}_f\mathcal{P}_f)$-coalgebra $[\![-]\!]_p \colon At \to \mathcal{C}(\mathcal{P}_f\mathcal{P}_f)(At)$.

Construction 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 1 as follows:*

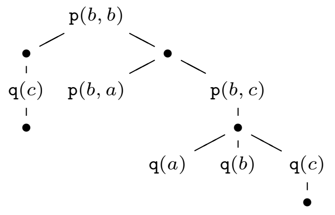
$$p_\alpha := \begin{cases} id_{At} & \alpha = 0 \\ \langle id_{At}, (\mathcal{P}_f\mathcal{P}_f(p_\beta) \circ p)\rangle & \alpha = \beta + 1. \end{cases}$$

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 $[\![-]\!]_p \colon At \to \mathcal{C}(\mathcal{P}_f\mathcal{P}_f)(At)$.

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 and-or tree up to depth m. As shown in [23], the limit $[\![-]\!]_p$ of all such approximations provides the full and-or tree of A.

Example 1. Consider the ground logic program on the left-hand side, based on an alphabet consisting of a signature $\{a^0, b^0, c^0\}$ and predicates $\mathtt{p}(-,-)$, $\mathtt{q}(-)$. The and-or tree $[\![\mathtt{p}(b,b)]\!]_p \in \mathcal{C}(\mathcal{P}_f\mathcal{P}_f)(At)$ is depicted on the right-hand side.

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



2.2 The General Case

We recall the extension of the coalgebraic semantics to arbitrary (i.e. possibly non-ground) logic programs presented in [25, 24]. In presence of variables, and-or trees are not guaranteed to represent sound derivations, whence *coinductive trees* are introduced as a sound variant of and-or trees, where unification is restricted to term-matching. We refer to [25, 24] and Appendix A for more details.

Before formally defining coinductive trees, it is worth recalling that, in [25], the collection of atoms (based on an alphabet $\mathcal{A}$) is modeled as a presheaf $At \colon \mathbf{L}^{op}_{\mathbf{\Sigma}} \to \mathbf{Set}$. The index category is the (opposite) *Lawvere Theory* $\mathbf{L}^{op}_{\mathbf{\Sigma}}$ of Σ, as defined above. For each natural number $n \in |\mathbf{L}^{op}_{\mathbf{\Sigma}}|$, $At(n)$ is defined as the set of atoms with variables among $x_1, \ldots, x_n$. Given an arrow $\theta \in \mathbf{L}^{op}_{\mathbf{\Sigma}}[n,m]$, 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 2. *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 1 and property 5 replaced by the following*[1]*:*

5. *For every and-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 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 [25] 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 2 suggests how p should act on an atom $A \in At(n)$, for a fixed n:

$$\begin{aligned} A \mapsto \{\{B_1, \ldots, B_k\}\tau \mid & \ H \leftarrow B_1, \ldots, B_k \text{ is a clause of } \mathbb{P}, \\ & A = H\tau \text{ and } B_1\tau, \ldots, B_k\tau \in At(n)\}. \end{aligned} \tag{1}$$

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. [25]). Thus the object on the right-hand side of (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 $At \colon \mathbf{L}_{\mathbf{\Sigma}}^{op} \to \mathbf{Set}$, consider liftings $\widetilde{\mathcal{P}_c} \colon \mathbf{Set}^{\mathbf{L}_{\mathbf{\Sigma}}^{op}} \to \mathbf{Set}^{\mathbf{L}_{\mathbf{\Sigma}}^{op}}$ and $\widetilde{\mathcal{P}_f} \colon \mathbf{Set}^{\mathbf{L}_{\mathbf{\Sigma}}^{op}} \to \mathbf{Set}^{\mathbf{L}_{\mathbf{\Sigma}}^{op}}$, standardly defined on presheaves $\mathcal{F} \colon \mathbf{L}_{\mathbf{\Sigma}}^{op} \to \mathbf{Set}$ by postcomposition respectively with $\mathcal{P}_c$ and $\mathcal{P}_f$. Then one would like to fix (1) as the definition of the n-component of a natural transformation $p \colon At \to \widetilde{\mathcal{P}_c}\widetilde{\mathcal{P}_f}(At)$. The key problem with this formulation is that p would *not* be a natural transformation, as shown by the following example.

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

$$\begin{array}{rcl@{\qquad\qquad}rl} \texttt{List}(cons(x_1, x_2)) & \leftarrow & \texttt{Nat}(x_1), \texttt{List}(x_2) & \texttt{List}(nil) & \leftarrow \\ \texttt{Nat}(succ(x_1)) & \leftarrow & \texttt{Nat}(x_1) & \texttt{Nat}(zero) & \leftarrow \end{array}$$

Fix a substitution $\theta = \langle nil \rangle \colon 1 \to 0$ and, for each $n \in |\mathbf{L}_{\mathbf{\Sigma}}^{op}|$, suppose that $p(n) \colon At(n) \to \widetilde{\mathcal{P}_c}\widetilde{\mathcal{P}_f}(At)(n)$ is defined according to (1). Then the square

[1] The notion of coinductive tree that we define here, differently from the one given in [24, Def.4.1], allows at a given depth multiple or-nodes to represent the same clause (with different term-matchers). In fact, it corresponds to the notion of coinductive forest of breadth n [24, Def.4.4], the only difference being that we "glue" together all trees of the forest into a single tree. This formulation is more convenient for our presentation. Observe that the adequacy theorem for the coalgebraic semantics [24, Th.4.5] is formulated in terms of coinductive forests and their breadth, whence one can also express it in terms of our notion of n-coinductive tree.

$$\begin{tikzcd}
At(1) \arrow[r, "p(1)"] \arrow[d, "At(\theta)"'] & \widetilde{\mathcal{P}_c}\widetilde{\mathcal{P}_f}(At)(1) \arrow[d, "\widetilde{\mathcal{P}_c}\widetilde{\mathcal{P}_f}(At)(\theta)"] \\
At(0) \arrow[r, "p(0)"'] & \widetilde{\mathcal{P}_c}\widetilde{\mathcal{P}_f}(At)(0)
\end{tikzcd}$$

does not commute. A counterexample is provided by the atom $\texttt{List}(x_1) \in At(1)$. Passing through the bottom-left corner of the square, $\texttt{List}(x_1)$ is mapped first to $\texttt{List}(nil) \in At(0)$ and then to $\{\emptyset\} \in \widetilde{\mathcal{P}_c}\widetilde{\mathcal{P}_f}(At)(0)$ - intuitively, this yields a refutation of the goal $\{\texttt{List}(x_1)\}$ with substitution of x_1 with nil. Passing through the top-right corner, $\texttt{List}(x_1)$ is mapped first to $\emptyset \in \widetilde{\mathcal{P}_c}\widetilde{\mathcal{P}_f}(At)(1)$ and then to $\emptyset \in \widetilde{\mathcal{P}_c}\widetilde{\mathcal{P}_f}(At)(0)$, i.e. the computation ends up in a failure.

In [25, Sec.4] the authors overcome this difficulty by relaxing the naturality requirement. The morphism p is defined as a $\breve{\mathcal{P}_c}\breve{\mathcal{P}_f}$-coalgebra in the category $Lax(\mathbf{L}_{\mathbf{\Sigma}}^{op}, \mathbf{Poset})$ of locally ordered functors $\mathcal{F}\colon \mathbf{L}_{\mathbf{\Sigma}}^{op} \to \mathbf{Poset}$ and *lax* natural transformations, with each component $p(n)$ given according to (1) and $\breve{\mathcal{P}_c}\breve{\mathcal{P}_f}$ the extension of $\widetilde{\mathcal{P}_c}\widetilde{\mathcal{P}_f}$ to an endofunctor on $Lax(\mathbf{L}_{\mathbf{\Sigma}}^{op}, \mathbf{Poset})$.

The lax approach fixes the problem, but presents also some drawbacks. Unlike the categories **Set** and $\mathbf{Set}^{\mathbf{L}_{\mathbf{\Sigma}}^{op}}$, $Lax(\mathbf{L}_{\mathbf{\Sigma}}^{op}, \mathbf{Poset})$ is neither complete nor cocomplete, meaning that a cofree comonad on $\breve{\mathcal{P}_c}\breve{\mathcal{P}_f}$ cannot be retrieved through the standard Constructions 1 and 2 that were used in the ground case. Moreover, the category of $\breve{\mathcal{P}_c}\breve{\mathcal{P}_f}$-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.

3 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.

$$\begin{tikzcd}
\mathbf{Set}^{\mathbf{L}_{\mathbf{\Sigma}}^{op}} \arrow[r, bend left, "\mathcal{U}"] \arrow[r, phantom, "\perp"] & \mathbf{Set}^{|\mathbf{L}_{\mathbf{\Sigma}}^{op}|} \arrow[l, bend left, "\mathcal{K}"]
\end{tikzcd}
\qquad
\begin{tikzcd}
{|\mathbf{L}_{\mathbf{\Sigma}}^{op}|} \arrow[r, hook, "\iota"] \arrow[d, "\mathcal{F}"'] & \mathbf{L}_{\mathbf{\Sigma}}^{op} \arrow[dl, "\mathcal{K}(\mathcal{F})"] \\
\mathbf{Set} &
\end{tikzcd}$$

The left adjoint $\mathcal{U}$ is the forgetful functor, given by precomposition with the inclusion functor $\iota\colon |\mathbf{L}_{\mathbf{\Sigma}}^{op}| \hookrightarrow \mathbf{L}_{\mathbf{\Sigma}}^{op}$. As shown in [29, Th.X.1], $\mathcal{U}$ has a right adjoint

$\mathcal{K}\colon \mathbf{Set}^{|\mathbf{L}_{\mathbf{\Sigma}}^{op}|} \to \mathbf{Set}^{\mathbf{L}_{\mathbf{\Sigma}}^{op}}$ sending $\mathcal{F}\colon |\mathbf{L}_{\mathbf{\Sigma}}^{op}| \to \mathbf{Set}$ to its *right Kan extension* along ι. This is a presheaf $\mathcal{K}(\mathcal{F})\colon \mathbf{L}_{\mathbf{\Sigma}}^{op} \to \mathbf{Set}$ mapping an object n of $\mathbf{L}_{\mathbf{\Sigma}}^{op}$ to

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

where m is any object of $\mathbf{L}_{\mathbf{\Sigma}}^{op}$. 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 $\mathcal{K}(\mathcal{F})\colon \mathbf{L}_{\mathbf{\Sigma}}^{op} \to \mathbf{Set}$ on an arrow $\theta\colon n \to m$ as

$$\mathcal{K}(\mathcal{F})(\theta)\colon \dot{x} \mapsto \langle \dot{x}(\sigma \circ \theta)\rangle_{\sigma: m \to m'}. \tag{2}$$

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

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

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

For any presheaf $\mathcal{F}\colon \mathbf{L}_{\mathbf{\Sigma}}^{op} \to \mathbf{Set}$, 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 $n \in |\mathbf{L}_{\mathbf{\Sigma}}^{op}|$, $X \in \mathcal{F}(n)$,

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

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 $\theta \in \mathbf{L}_{\mathbf{\Sigma}}^{op}[n, m]$.

As shown in Example 2, given a program $\mathbb{P}$, the family of functions p defined by (1) fails to be a morphism in $\mathbf{Set}^{\mathbf{L}_{\mathbf{\Sigma}}^{op}}$. However, it forms a morphism in $\mathbf{Set}^{|\mathbf{L}_{\mathbf{\Sigma}}^{op}|}$

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

where $\widehat{\mathcal{P}_c}$ and $\widehat{\mathcal{P}_f}$ denote the liftings of $\mathcal{P}_c$ and $\mathcal{P}_f$ to $\mathbf{Set}^{|\mathbf{L}_{\mathbf{\Sigma}}^{op}|}$. The naturality requirement is trivially satisfied in $\mathbf{Set}^{|\mathbf{L}_{\mathbf{\Sigma}}^{op}|}$, since $|\mathbf{L}_{\mathbf{\Sigma}}^{op}|$ 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 $\mathbf{Set}^{\mathbf{L}_{\mathbf{\Sigma}}^{op}}$, defined as

$$At \xrightarrow{\eta_{At}} \mathcal{K}\mathcal{U}(At) \xrightarrow{\mathcal{K}(p)} \mathcal{K}\widehat{\mathcal{P}_c}\widehat{\mathcal{P}_f}\mathcal{U}(At). \tag{3}$$

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 $\langle p(m)(A\sigma)\rangle_{\sigma: n \to m}$, that is:

$$\begin{aligned} p^\sharp(n)\colon A \mapsto \langle \{\{B_1, \ldots, B_k\}\tau \mid H \leftarrow B_1, \ldots, B_k \text{ is a clause of } \mathbb{P}, \\ A\sigma = H\tau \text{ and } B_1\tau, \ldots, B_k\tau \in At(m)\}\rangle_{\sigma: n \to m}. \end{aligned} \tag{4}$$

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 2.

Example 3. Consider the same square of Example 2, with $p^\sharp$ in place of p and $\mathcal{S}$ in place of $\widetilde{\mathcal{P}_c\mathcal{P}_f}$. 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 (2). 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 the atom $\mathtt{List}(x_1)$.

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 **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 1 and 2.

Construction 3 *The terminal sequence for the functor* $At \times \mathcal{S}(-)\colon \mathbf{Set}^{\mathbf{L}_\Sigma^{op}} \to \mathbf{Set}^{\mathbf{L}_\Sigma^{op}}$ *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 1, with* $\mathcal{S}$ *replacing* $\mathcal{P}_f\mathcal{P}_f$*. As shown in Appendix B, 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, the *cofree comonad* $\mathcal{C}(\mathcal{S})$ exists and maps At to X_γ given as in Construction 3. A $\mathcal{C}(\mathcal{S})$-coalgebra $[\![-]\!]_{p^\sharp}\colon At \to \mathcal{C}(\mathcal{S})(At)$ is given below.

Construction 4 *The terminal sequence for* $At\times\mathcal{S}(-)$ *induces a cone* $\{p^\sharp_\alpha\colon At \to X_\alpha\}_{\alpha<\gamma}$ *as in Construction 2 with* $p^\sharp$ *and* $\mathcal{S}$ *replacing* p *and* $\mathcal{P}_f\mathcal{P}_f$*. This yields a natural transformation* $[\![-]\!]_{p^\sharp}\colon At \to X_\gamma$*, where* $X_\gamma = \mathcal{C}(\mathcal{S})(At)$*.*

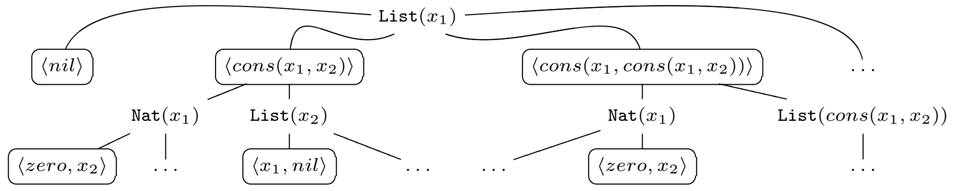
As in the ground case, the coalgebra $[\![-]\!]_{p^\sharp}$ is constructed as an iterative application of $p^\sharp$: we call *saturated tree* the associated tree structure.

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

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

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

Example 4. Part of the infinite saturated tree of $\mathtt{List}(x_1) \in At(1)$ in $\mathtt{NatList}$ is depicted below. Note that not all labels of and-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 or-node, which is now a substitution. For instance, both $\mathtt{Nat}(x_1)$ and $\mathtt{List}(x_2)$ 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).



We can generalize these observations to the following adequacy theorem.

Theorem 1. *Let $[\![-]\!]_{p^\sharp}$ be defined from a program $\mathbb{P}$ according to Construction 4. Then, for all n and $A \in At(n)$, the saturated tree of A in $\mathbb{P}$ is $[\![A]\!]_{p^\sharp}$.*

In the above theorem and in the rest of the paper, with an abuse of notation we use $[\![A]\!]_{p^\sharp}$ to denote the application of $[\![-]\!]_{p^\sharp}(n)$ to $A \in At(n)$ without mentioning the object $n \in |\mathbf{L}^{op}_{\mathbf{\Sigma}}|$. For an arrow $\theta \in \mathbf{L}^{op}_{\mathbf{\Sigma}}[n, m]$, 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 following theorem that is an immediate consequence of the naturality of $[\![-]\!]_{p^\sharp}$.

Theorem 2 (Compositionality). *For all atoms $A \in At(n)$ and substitutions $\theta \in \mathbf{L}^{op}_{\mathbf{\Sigma}}[n, m]$,*

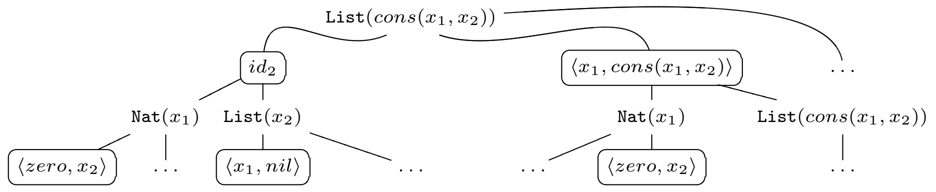
$$[\![A\theta]\!]_{p^\sharp} = [\![A]\!]_{p^\sharp}\overline{\theta}.$$

We conclude this section with a concrete description of the behavior of the operator $\overline{\theta}$, for a given substitution $\theta \in \mathbf{L}^{op}_{\mathbf{\Sigma}}[n, m]$. 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 or-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 (2), 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 3.

Example 5. Recall from Example 4 the saturated tree $[\![\mathtt{List}(x_1)]\!]_{p^\sharp}$. For $\theta = \langle cons(x_1, x_2)\rangle$, the tree $[\![\mathtt{List}(x_1)]\!]_{p^\sharp}\overline{\theta}$ is depicted below.



4 Desaturation

One of the main features of coinductive trees is to represent (sound) and-or parallel derivations of goals. This leads the authors of [24] to a resolution algorithm exploiting the two forms of parallelism [26]. Motivated by these developments, we include coinductive trees in our framework, showing how they can be obtained as a "desaturation" of saturated trees.

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

$$\epsilon_{\mathcal{F}}(n) \colon \dot{x} \mapsto \dot{x}(id_n) \tag{5}$$

where $\dot{x}(id_n)$ is the element of the input tuple $\dot{x}$ which is indexed by the identity substitution $id_n \in \mathbf{L}_{\mathbf{\Sigma}}^{op}[n,n]$. 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.

The next construction defines a morphism $\overline{d} \colon \mathcal{U}\big(\mathcal{C}(\mathcal{S})(At)\big) \to \mathcal{C}(\widehat{\mathcal{P}_c}\widehat{\mathcal{P}_f})(\mathcal{U}At)$ where $\mathcal{C}(\widehat{\mathcal{P}_c}\widehat{\mathcal{P}_f}) \colon \mathbf{Set}^{|\mathbf{L}_{\mathbf{\Sigma}}^{op}|} \to \mathbf{Set}^{|\mathbf{L}_{\mathbf{\Sigma}}^{op}|}$ is the cofree comonad on $\widehat{\mathcal{P}_c}\widehat{\mathcal{P}_f}$, obtained through a terminal sequence analogously to Construction 3. The idea is that $\overline{d}$ acts on saturated trees as the depthwise application of $\epsilon_{\mathcal{U}At}$.

Construction 5 *For α an ordinal, let us note by Y_α the objects occurring in the construction of $\mathcal{C}(\widehat{\mathcal{P}_c}\widehat{\mathcal{P}_f})(\mathcal{U}At)$ and with X_α the ones in the construction of $\mathcal{C}(\mathcal{S})(At)$, converging to $X_\gamma = \mathcal{C}(\mathcal{S})(At)$. We define a sequence $\{d_\alpha \colon \mathcal{U}(X_\alpha) \to Y_\alpha\}_{\alpha<\gamma}$ in $\mathbf{Set}^{|\mathbf{L}_{\mathbf{\Sigma}}^{op}|}$ as follows:*

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

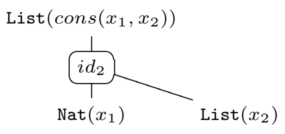
For $\alpha < \gamma$ a limit ordinal, $d_\alpha \colon \mathcal{U}(X_\alpha) \to Y_\alpha$ is provided by the limiting property of Y_α. This sequence induces a morphism $\overline{d} \colon \mathcal{U}\big(\mathcal{C}(\mathcal{S})(At)\big) \to \mathcal{C}(\widehat{\mathcal{P}_c}\widehat{\mathcal{P}_f})(\mathcal{U}At)$. We refer to Appendix C for more details.

The next theorem shows 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 trees where the labeling of or-nodes has been restricted to the identity substitution id_n, represented as $\bullet$ (see Definition 2). The operation of pruning all or-nodes (and their descendants) in $[\![A]\!]_{p^\sharp}$ which are not labeled with id_n is precisely what is provided by Construction 5, in virtue of the definition of the counit ϵ given in (5).

Theorem 3 (Desaturation). *Let $[\![-]\!]_{p^\sharp} \colon At \to \mathcal{C}(\mathcal{S})(At)$ be defined for a logic program $\mathbb{P}$ according to Construction 4 and $\overline{d} \colon \mathcal{U}\big(\mathcal{C}(\mathcal{S})(At)\big) \to \mathcal{C}(\widehat{\mathcal{P}_c}\widehat{\mathcal{P}_f})(\mathcal{U}At)$ be defined according to Construction 5. Then for all $n \in |\mathbf{L}_{\mathbf{\Sigma}}^{op}|$ and $A \in \mathcal{U}At(n)$, the n-coinductive tree of A in $\mathbb{P}$ is $\big(\overline{d} \circ \mathcal{U}([\![-]\!]_{p^\sharp})\big)(n)(A)$.*

Theorem 3 also provides an alternative formalization for the coinductive tree semantics [25], 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 $\mathbf{L}_{\mathbf{\Sigma}}^{op}$, shifting the framework from $\mathbf{Set}^{\mathbf{L}_{\mathbf{\Sigma}}^{op}}$ to $\mathbf{Set}^{|\mathbf{L}_{\mathbf{\Sigma}}^{op}|}$. The substitutions on trees (that are essential, for instance, for the resolution algorithm given in [24, 26]) 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 3.

Example 6. 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 5, i.e., by pruning all the or-nodes (and their descendants) that are not labeled with id_2.



5 Soundness and Completeness

The notion of coinductive tree leads to a semantics that is sound and complete with respect to SLD-resolution [24, Th.4.8]. To this aim, a key role is played by *derivation subtrees* of a given coinductive tree.

Definition 4. *Let T be the n-coinductive 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 and-node of T belongs to T', then just one of its children belongs to T';*
3. *if an or-node of T belongs to T', then all its children belong to T'.*

A refutation subtree *(called success subtree in [24]) is a finite derivation subtree with only or-nodes as leaves.*

In analogy with coinductive trees, we want to define a notion of subtree for saturated semantics. This requires care: saturated trees are associated with unification, which is more liberal than term-matching. In particular, similarly to and-or trees, they may represent unsound derivation strategies. However, in saturated trees *all* unifiers, and 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 5. *Let T be the saturated 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 4 and the following condition:*

4. *all or-nodes of T' which are at the same depth are labeled with the same substitution.*

A synched refutation subtree *is a finite synched derivation subtree with only or-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 or-nodes of depth i and $2k+1$ is its maximal depth.*

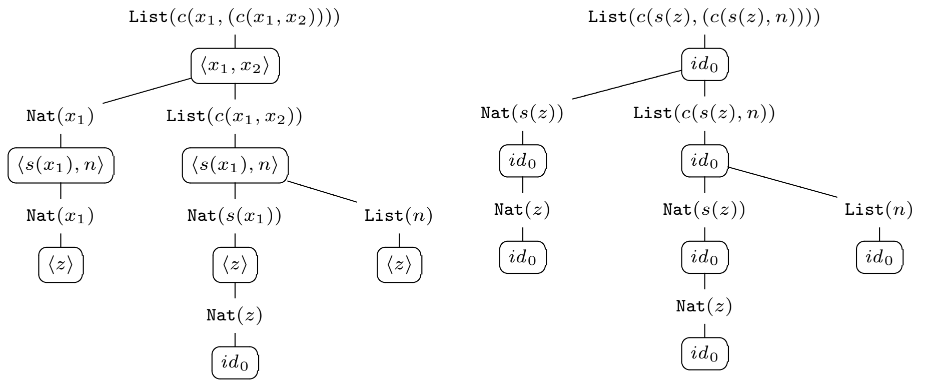


Fig. 1. Successful synched derivation subtrees for $\mathtt{List}(\mathit{cons}(x_1,(\mathit{cons}(x_1,x_2))))$ (left) and $\mathtt{List}(\mathit{cons}(\mathit{succ}(\mathit{zero}),(\mathit{cons}(\mathit{succ}(\mathit{zero}),\mathit{nil}))))$ (right) in $\mathtt{NatList}$. The symbols *cons*, *nil*, *succ* and *zero* are abbreviated to c, n, s and z respectively.

The prefix "synched" emphasizes the restriction to and-parallelism which is encoded in Definition 5. Intuitively, we force all subgoals at the same depth to proceed with *the same* substitution. For instance, this rules out the unsound derivation of [24, Ex.5.2] (reported in Appendix A, Example 7).

Note that derivation subtrees can be seen as special instances of synched derivation subtrees where all the substitutions are forced to be identities.

Theorem 4 (Soundness and Completeness). *Let $\mathbb{P}$ be a logic program and $A \in At(n)$ an atom. The following are equivalent.*

1. *The saturated tree for A in $\mathbb{P}$ has a synched refutation subtree with answer θ.*
2. *There is some natural number m such that 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 computed answer τ such that there exists a substitution σ with $\sigma \circ \tau = \theta$.*

The statement $(2 \Leftrightarrow 3)$ is a rephrasing of [24, Th.4.8], while $(1 \Leftrightarrow 2)$ is proved in Appendix D by using compositionality and desaturation (Theorems 2 and 3).

Figure 1 provides an example of the argument for direction $(1 \Rightarrow 2)$. 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 key observation is that the rightmost tree is a refutation subtree of the 0-coinductive tree for $A\theta$ and can be obtained from the leftmost tree by a procedure involving the operator $\overline{\theta}$ discussed at the end of Section 3.

6 Conclusions

This work proposed a coalgebraic semantics for logic programming, extending the framework introduced in [23] 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. We stressed how the critical point of this generalization was to achieve compositionality, which we obtained by employing *saturation* techniques. Starting from the operational semantics p proposed in [25], we characterized its saturation $p^\sharp$ in terms of substitution mechanisms, showing that the latter corresponds to unification, whereas the former is associated with term-matching. The map $p^\sharp$ gave rise to the notion of *saturated tree*, as the model of computation represented in our semantics. We observed that coinductive trees, introduced in [25], can be seen as a desaturated version of saturated trees, and we compared the two notions with a translation. Eventually, we tailored a notion of subtree (of a saturated 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. A next step in this direction would be to investigate *infinite* computations and the semantics of coinductive logic programming [20]. These have been fruitfully explored within the approach based on coinductive trees [24, 26], and we expect the notion of synched derivation subtree 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 [7] 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, since minimal arrows have a very intuitive characterization as most general unifiers.

References

1. J. Adámek and V. Koubek. On the greatest fixed point of a set functor. *Theor. Comput. Sci.*, 150:57–75, 1995.
2. J. Adámek and J. Rosický. *Locally Presentable and Accessible Categories.* Cambridge University Press, 1994.
3. G. Amato, J. Lipton, and R. McGrail. On the algebraic structure of declarative programming languages. *Theor. Comput. Sci.*, 410(46):4626–4671, 2009.
4. F. Bonchi. *Abstract Semantics by Observable Contexts.* PhD thesis, Department of Computer Science of Pisa, 2008.
5. 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.
6. F. Bonchi, B. König, and U. Montanari. Saturated semantics for reactive systems. In *LICS*, pages 69–80. IEEE, 2006.
7. F. Bonchi and U. Montanari. Coalgebraic symbolic semantics. In *CALCO*, pages 173–190, 2009.
8. F. Bonchi and U. Montanari. Reactive systems, (semi-)saturated semantics and coalgebras on presheaves. *Theor. Comput. Sci.*, 410(41):4044–4066, 2009.
9. R. Bruni, U. Montanari, and F. Rossi. An interactive semantics of logic programming. *Theory and Practice of Logic Programming*, 1(6):647–690, 2001.
10. A. Corradini, M. Große-Rhode, and R. Heckel. A coalgebraic presentation of structured transition systems. *Theor. Comput. Sci.*, 260(1-2):27–55, 2001.
11. 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.

12. 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.
13. 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.
14. M. P. Fiore, E. Moggi, and D. Sangiorgi. A fully abstract model for the π-calculus. *Inf. Comput.*, 179(1):76–117, 2002.
15. M. P. Fiore and S. Staton. Comparing operational models of name-passing process calculi. *Inf. Comput.*, 204(4):524–560, 2006.
16. 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.
17. M. P. Fiore and D. Turi. Semantics of name and value passing. In *LICS*, pages 93–104. IEEE, 2001.
18. N. Ghani, K. Yemane, and B. Victor. Relationally staged computations in calculi of mobile processes. volume 106, pages 105–120, 2004.
19. 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.
20. G. Gupta, A. Bansal, R. Min, L. Simon, and A. Mallya. Coinductive logic programming and its applications. In *ICLP*, pages 27–44, 2007.
21. G. Gupta and V. S. Costa. Optimal implementation of and-or parallel prolog. In *PARLE*, pages 71–92, 1994.
22. Y. Kinoshita and A. J. Power. A fibrational semantics for logic programs. In *Proceedings of the 5th International Workshop on Extensions of Logic Programming*, ELP '96, pages 177–191, London, UK, UK, 1996. Springer-Verlag.
23. E. Komendantskaya, G. McCusker, and J. Power. Coalgebraic semantics for parallel derivation strategies in logic programming. In *AMAST*, pages 111–127, 2010.
24. E. Komendantskaya and J. Power. Coalgebraic derivations in logic programming. In *CSL*, pages 352–366, 2011.
25. E. Komendantskaya and J. Power. Coalgebraic semantics for derivations in logic programming. In *CALCO*, pages 268–282, 2011.
26. E. Komendantskaya, J. Power, and M. Schmidt. Coalgebraic logic programming: from semantics to implementation. *Submitted to the Journal of Logic and Computation*, 2012.
27. R. Kowalski. *Logic for problem-solving.* North-Holland Publishing Co., Amsterdam, The Netherlands, The Netherlands, 1986.
28. J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2nd edition, 1993.
29. S. Mac Lane. *Categories for the Working Mathematician.* Graduate Texts in Mathematics. Springer, 2nd edition, Sept. 1998.
30. S. MacLane and I. Moerdijk. *Sheaves in Geometry and Logic: A First Introduction to Topos Theory.* Springer, corrected edition, May 1992.
31. Z. Majkic. Coalgebraic semantics for logic programs. In *Proceedings of the 18th Workshop W(C)LP on Constraint Logic Programming*, pages 76–87, 2004.
32. M. Miculan. A categorical model of the fusion calculus. *ENTCS*, 218:275–293, 2008.
33. M. Miculan and K. Yemane. A unifying model of variables and names. In *FOSSACS*, volume 3441, pages 170–186, 2005.
34. U. Montanari and M. Sammartino. A network-conscious pi-calculus and its coalgebraic semantics. *Submitted to TCS (Festschrift for Glynn Winskel).*
35. U. Montanari and V. Sassone. Dynamic congruence vs. progressing bisimulation for ccs. *FI*, 16(1):171–199, 1992.
36. D. Sangiorgi. A theory of bisimulation for the pi-calculus. *Acta Inf.*, 33(1):69–97, 1996.
37. I. Stark. A fully abstract domain model for the π-calculus. In *LICS*, pages 36–42. IEEE, 1996.
38. S. Staton. Relating coalgebraic notions of bisimulation. In *CALCO*, pages 191–205, 2009.
39. J. Worrell. Terminal sequences for accessible endofunctors. *ENTCS*, 19, 1999.

A More on SLD-Resolution and And-Or Trees

In this appendix we provide a more detailed description of SLD-resolution and the problem of unsoundness in and-or trees.

As mentioned in Section 2, SLD-resolution is an algorithm to check whether a goal G (which we represent as a finite set of atoms) is *refutable* in a program $\mathbb{P}$. 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

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

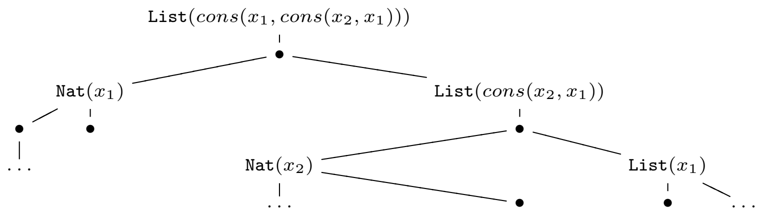
Such a computation is called an *SLD-refutation* if it terminates in a finite number (say n) of steps with $G_n = \emptyset$. In this case one calls *computed answer* the substitution given by composing the first projections $\sigma_n, \ldots, \sigma_0$ of the most general unifiers associated with each step of the computation. The goal G is *refutable* in $\mathbb{P}$ if an SLD-refutation for G in $\mathbb{P}$ exists[2].

As reported in Section 2, implementations of SLD-resolution exploiting *and-or parallelism* have a standard representation of computations as *and-or trees* (Definition 1). Unfortunately, in presence of variables these 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 [13].

Example 7. Let `NatList` be the same program of Example 2 and consider the atom $A = \texttt{List}(cons(x_1, cons(x_2, x_1)))$. 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 `NatList`. However, consider the and-or tree of A in

[2] 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) 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 for instance [19]. The different target corresponds to the two substitutions depending on disjoint sets of variables.

NatList, for which we provide a partial representation as follows.



The above tree seems to yield an SLD-refutation: List($cons(x_1, cons(x_2, x_1))$) is refuted by proving Nat(x_1) and 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 Nat(x_1) maps x_1 to *zero* and the computed answer of List($cons(x_2, x_1)$) maps x_1 to *nil*).

In Section 2 we recall *coinductive trees* [25]: a variant of and-or trees where unification is restricted to the case of 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.

For instance, the coinductive tree for the atom A of Example 7 in NatList is the subtree of the represented and-or tree having the nodes labeled with Nat(x_1), Nat(x_2) and List(x_1) as leaves. Despite of the restriction to term-matching, one can show correctness and completeness of coinductive tree semantics with respect to SLD-resolution, as reported in Theorem 4 and originally shown in [24].

B Convergence of a Terminal Sequence

This appendix is devoted to the following proposition, which is propaedeutic to Construction 3.

Proposition 1. *The terminal sequence for $At \times \mathcal{S}(-)$ converges to a terminal coalgebra.*

Proof. By [39, Th.7], 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 [2, 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.
– The functors $\widehat{\mathcal{P}_c} \colon \mathbf{Set}^{|\mathbf{L}_{\mathbf{\Sigma}}^{op}|} \to \mathbf{Set}^{|\mathbf{L}_{\mathbf{\Sigma}}^{op}|}$ and $\widehat{\mathcal{P}_f} \colon \mathbf{Set}^{|\mathbf{L}_{\mathbf{\Sigma}}^{op}|} \to \mathbf{Set}^{|\mathbf{L}_{\mathbf{\Sigma}}^{op}|}$ are defined as liftings of $\mathcal{P}_c \colon \mathbf{Set} \to \mathbf{Set}$ and $\mathcal{P}_f \colon \mathbf{Set} \to \mathbf{Set}$. It is well-known that $\mathcal{P}_c$ and $\mathcal{P}_f$ 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 [30, Ch.6]).

C A Closer Look at Desaturation

In this appendix we spell out the details of the construction of $\overline{d}$, as presented in Section 4. For this purpose, first we state the construction of the cofree comonad on $\widehat{\mathcal{P}}_c\widehat{\mathcal{P}}_f$ for later reference.

Construction 6 *The terminal sequence for $\mathcal{U}At \times \widehat{\mathcal{P}}_c\widehat{\mathcal{P}}_f(-) \colon \mathbf{Set}^{|\mathbf{L}_\Sigma^{op}|} \to \mathbf{Set}^{|\mathbf{L}_\Sigma^{op}|}$ consists of sequences of objects Y_α and arrows $\lambda_\alpha \colon Y_{\alpha+1} \to Y_\alpha$, defined by induction on α as follows.*

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

For α a limit ordinal, Y_α and λ_α are defined as expected. As stated in the proof of Proposition 1, $\widehat{\mathcal{P}}_c\widehat{\mathcal{P}}_f$ is a mono-preserving accessible functors. Then by [39, 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 $\mathcal{C}(\widehat{\mathcal{P}}_c\widehat{\mathcal{P}}_f) \colon \mathbf{Set}^{|\mathbf{L}_\Sigma^{op}|} \to \mathbf{Set}^{|\mathbf{L}_\Sigma^{op}|}$ on $\mathcal{U}At$, where $\mathcal{C}(\widehat{\mathcal{P}}_c\widehat{\mathcal{P}}_f)$ is the cofree comonad on $\widehat{\mathcal{P}}_c\widehat{\mathcal{P}}_f$ induced by the terminal sequence given above, analogously to Construction 1.

Now, we can define the desaturation map $\overline{d} \colon \mathcal{U}\big(\mathcal{C}(\mathcal{S})(At)\big) \to \mathcal{C}(\widehat{\mathcal{P}}_c\widehat{\mathcal{P}}_f)(\mathcal{U}At)$, filling in the details of Construction 5.

Construction 7 *Consider the image of the terminal sequence converging to $\mathcal{C}(\mathcal{S})(At) = X_\gamma$ (Construction 3) under the forgetful functor $\mathcal{U} \colon \mathbf{Set}^{\mathbf{L}_\Sigma^{op}} \to \mathbf{Set}^{|\mathbf{L}_\Sigma^{op}|}$. We define a sequence of natural transformations $\{d_\alpha \colon \mathcal{U}(X_\alpha) \to Y_\alpha\}_{\alpha<\gamma}$ as follows[3]:*

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

$$\begin{tikzcd}
\mathcal{U}(X_\beta) \arrow[rr, "d_\beta"] & & Y_\beta \\
\mathcal{U}(X_{\beta+1}) \arrow[u, "\mathcal{U}(\delta_\beta)"] \arrow[rr, "d_{\beta+1}"] \arrow[dr, "id_{\mathcal{U}At} \times \epsilon_{\widehat{\mathcal{P}}_c\widehat{\mathcal{P}}_f\mathcal{U}(X_\beta)}"'] & & Y_{\beta+1} \arrow[u, "\lambda_\beta"'] \\
& \mathcal{U}At \times \widehat{\mathcal{P}}_c\widehat{\mathcal{P}}_f\mathcal{U}(X_\beta) \arrow[ur, "id_{\mathcal{U}At} \times \widehat{\mathcal{P}}_c\widehat{\mathcal{P}}_f(d_\beta)"'] &
\end{tikzcd}$$

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

For $\alpha < \gamma$ a limit ordinal, a natural transformation $d_\alpha \colon \mathcal{U}(X_\alpha) \to Y_\alpha$ 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, $\lambda_\beta \circ d_{\beta+1} = d_\beta \circ \mathcal{U}(\delta_\beta)$. This can be easily checked by ordinal induction, using the fact that $\epsilon_{\widehat{\mathcal{P}_c}\widehat{\mathcal{P}_f}\mathcal{U}(X_\beta)}$ is a natural transformation, for each such β.

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 $\mathcal{U}(X_\chi)$.

D Proof of Theorem 4

In this appendix we provide more details on the proof of Theorem 4. To this aim, a key role is played by the following lemma that generalizes the example of Figure 1.

Lemma 1. *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 or-nodes are all labeled with id_m.*

Proof. First, we observe that the two following properties hold for all the and-nodes of $[\![A]\!]_{p^\sharp}$.

(†) Let θ, θ' be two arrows in $\mathbf{L}_{\mathbf{\Sigma}}^{op}$ such that $\theta' \circ \theta$ is defined. If an and-node s has a child t such that (a) the label of t is θ and (b) t has children labeled with $B_1, \dots, B_n$, 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 $B_1\theta', \dots, B_n\theta'$.

(‡) Let $\theta, \theta', \sigma, \sigma'$ be four arrows in $\mathbf{L}_{\mathbf{\Sigma}}^{op}$ such that $\sigma \circ \theta = \sigma' \circ \theta'$. If an and-node labeled with A' has a child t such that (a) the label of t is θ and (b) t has children labeled with $B_1, \dots, B_n$, 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 $B_1\sigma, \dots, B_n\sigma$.

Assume that $[\![A]\!]_{p^\sharp}$ has a synched refutation subtree T whose or-nodes are labeled with $\theta_1, \theta_3, \dots, \theta_{2k+1}$ (where θ_i is the substitution labeling the or-nodes of depth i). We prove that $[\![A]\!]_{p^\sharp}$ has another synched refutation subtree T' whose first or-node is labeled with $\theta = \theta_{2k+1} \circ \theta_{2k-1} \circ \dots \circ \theta_1$ and all the other or-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 $B_1^2 \dots B_{n_2}^2$. By (†), r has another child t' (in $[\![A]\!]_{p^\sharp}$), that (a) is labeled with θ and (b) has children labeled with $B_1^2\sigma_3 \dots B_n^2\sigma_3$ where $\sigma_3 = \theta_{k+1} \circ \theta_{k-1} \circ \dots \circ \theta_3$. 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 $i \leq 2k$, let σ_{i+1} denote $\theta_{2k+1} \circ \theta_{2k-1} \circ \cdots \circ \theta_{i+1}$ and let $B^i_1, \ldots, B^i_{n_i}$ be the labels of the and-nodes of T at depth i. The depth i of T' is given by and-nodes labeled with $B^i_1\sigma_{i+1}, \ldots, B^i_{n_i}\sigma_{i+1}$; the depth $i+1$ by or-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 $B^i_1, \ldots, B^i_{n_i}$ 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 $B^i_1\sigma_{i+1}, \ldots, B^i_{n_i}\sigma_{i+1}$ have children (in $[\![A]\!]_{p^\sharp}$) that (a) are labeled with id_m and (b) have children with labels $B^{i+2}_1\sigma_{i+3}, \ldots, B^{i+2}_{n_{i+2}}\sigma_{i+3}$.

Once we have built T', we can easily conclude. Recall that t' (the first or-node of T') is labeled with θ. Following the construction at the end of Section 3, 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.

We are now ready to provide a proof of Theorem 4.

Proof (Proof of Theorem 4). The statement ($2 \Leftrightarrow 3$) is a rephrasing of [24, Th.4.8], whence we focus on proving ($1 \Leftrightarrow 2$), where m is always the target object of θ.

($1 \Rightarrow 2$). If $[\![A]\!]_{p^\sharp}$ has a synched refutation subtree with answer θ, then by Lemma 1, $[\![A]\!]_{p^\sharp}\overline{\theta}$ has a synched refutation subtree T whose or-nodes are all labeled with id_m. Compositionality (Theorem 2) guarantees that T is a synched refutation subtree of $[\![A\theta]\!]_{p^\sharp}$. Since all the or-nodes of T are labeled with id_m, T is preserved by desaturation. This means that T is a refutation subtree of $\overline{d}(\mathcal{U}([\![-]\!]_{p^\sharp}))(m)(A\theta)$ which, by Theorem 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 3, this is also the case for $\overline{d}(\mathcal{U}([\![-]\!]_{p^\sharp}))(m)(A\theta)$. This means that T is a synched derivation subtree of $[\![A\theta]\!]_{p^\sharp}$ whose or-nodes are all labeled by id_m. By compositionality, T is also a subtree of $[\![A]\!]_{p^\sharp}\overline{\theta}$. Let t be the or-node at the first depth of T. By construction of the operator $\overline{\theta}$, the root of $[\![A]\!]_{p^\sharp}$ has a child t' labeled with θ having the same children as t. Therefore $[\![A]\!]_{p^\sharp}$ has a synched refutation subtree with answer θ.