Coalgebraic Semantics for Logic Programming

Ekaterina Komendantskaya[1,4], Guy McCusker[2], John Power[3,5]

School of Computer Science, University of St Andrews, United Kingdom
Department of Computer Science, University of Bath, United Kingdom

Abstract

Logic programming, a class of programming languages based on first-order logic, provides simple and efficient tools for goal-oriented proof-search. Logic programming supports recursive computations - and some logic programs resemble the inductive or coinductive definitions written in functional programming languages. In this paper, we give a coalgebraic semantics to logic programming. We show that ground logic programs - be it PROLOG language with function symbols or DATALOG language without functions - can be modelled by either $P_f P_f$-coalgebras or $P_f List$-coalgebras on Set. We analyse different kinds of derivation strategies and derivation trees (proof-trees, SLD-trees, and-or parallel trees) used in logic programming, and show how they can be modelled coalgebraically. We extend these models to non-ground logic programs by incorporating a *Lawvere theory* into this picture. Namely, a given first-order signature Σ generates the Lawvere theory $\mathcal{L}_\Sigma$, and the coalgebraic semantics of non-ground logic programs can be obtained by replacing Set by $Lax(\mathcal{L}_\Sigma, Poset)$ and by allowing countability.

Keywords: Logic programming, SLD-resolution, Lawvere Theory, Coalgebra, Coinduction.

1 Introduction

In the standard formulations of logic programming, such as in Lloyd's book [23], a logic program consists of a finite set of clauses of the form

$$A \leftarrow A_1, \ldots, A_n$$

where A and the A_i's are atomic formulae, typically containing free variables; and $A_1, \ldots, A_n$ is to mean the conjunction of the A_i's. Note that n may be 0. The central algorithm for logic programming, called SLD-resolution, takes a goal $G = \leftarrow B_1, \ldots, B_n$, which is also meant as a conjunction of atomic formulae typically containing free variables, and constructs a proof for an instantiation of G from

[1] Email: mailto:ek@cs.st-andrews.ac.uk
[2] Email: mailto:G.A.McCusker@bath.ac.uk
[3] Email: mailto:A.J.Power@bath.ac.uk
[4] This work is supported by EPSRC postdoctoral fellowship EP/F044046/1.
[5] This document is an output from the PMI2 Project funded by the UK Department for Innovation, Universities and Skills (DIUS) for the benefit of the Japanese Higher Education Sector and the UK Higher Education Sector. The views expressed are not necessarily those of DIUS, nor British Council

substitution instances of the clauses in a given logic program P. The algorithm uses Horn-clause logic, with variable substitution determined universally to make the first atom in G agree with the head of a clause in P, then proceeding inductively.

This situation cries out for investigation in terms of coalgebra. Observe that the implication $\leftarrow$ does not appear as a proper subformula of a clause. So $\leftarrow$ acts at a meta-level, like a sequent rather than a logical connective. In more sophisticated forms of logic programming, such as [26], $\leftarrow$ can act as both a sequent and a logical connective. But treatment of $\leftarrow$ as a sequent remains standard and we shall assume it in the paper.

For the moment, let us ignore variables: we shall reinstate them later. And assume there are only finitely many predicate symbols, as implicit in the definition of logic program. Then, an atomic formula A may be the head of one clause, or no clause, or many clauses, but only ever finitely many. Each clause with head A has a finite number of atomic formulae $A_1, \ldots, A_n$ in its antecedent. So one can see a logic program, without variables, as a coalgebra of the form

$$p : At \longrightarrow P_f(P_f(At))$$

where At is the set of atomic formulae and p sends an atomic formula A to the set of the set of atomic formulae in each antecedent of each clause for which A is the head.

Thus we can identify a variable-free logic program with a P_fP_f-coalgebra on Set. In fact, we can go further. If we let $C(P_fP_f)$ be the cofree comonad on P_fP_f, then given a logic program qua P_fP_f-coalgebra, the corresponding $C(P_fP_f)$-coalgebra structure describes the and-or parallel derivation trees of the logic program yielding the basic computational construct of logic programming in the variable-free setting. We explain this in Section 4.

A similar analysis of SLD-derivations in logic programming can be given in terms of P_fList-coalgebras. Such analysis would be suitable for logic programming applications that treat conjunctions $\leftarrow B_1, B_2, \ldots, B_n$ as lists. The main difference between the models based on P_fP_f- and P_fList-coalgebras, is that they model different strategies of parallel SLD-derivations.

The variable-free setting is more general than may first appear: because of finiteness, often only finitely many "ground" substitutions are possible; the exception to this are logic programs that describe potentially infinite data. Therefore, one often can emulate an arbitrary logic program by a variable-free one. Nevertheless, even in its own terms, we can in principle extend our analysis to arbitrary logic programming by replacing Set by the category $Set^{\mathcal{L}_\Sigma}$, where $\mathcal{L}_\Sigma$ is the Lawvere theory generated by the operations of a signature Σ. The Lawvere theory $\mathcal{L}_\Sigma$ has exactly those equalisers that correspond to "most general unifiers" in logic programming, cf. [32]. In detail, the situation is more complex than this. The main complications are that substitution can generate infinitely many clauses from a finite number of them, and it is possible that a clause only appears in a logic program after substitution. We discuss this in Section 5.

First-order logic programming in general is *P-complete*, and thus inherently sequential. Therefore, in non-ground cases, the models based on P_fP_f-coalgebras

would remain sound only for certain kinds of DATALOG programs - logic programs containing no function symbols, see [35,17]. We discuss this in Section 6.

The paper is organised as follows. In Section 2, we recall the fundamental concepts of logic programming. In Section 3, we discuss the operational semantics for logic programs given by the SLD resolution, in particular paying attention to inductive and coinductive aspects of logic programming. In Section 4, we show how to model variable-free logic programs as coalgebras and exhibit the role of the cofree comonad; we discuss the difference between SLD-refutations modelled by $P_f P_f$- and $P_f List$-coalgebras. In Section 5, we give semantics to first-order signature, and show that it corresponds to *Lawvere theory*. In Section 6, we use the Lawvere theory generated by the first-order signature to extend the models given by $P_f P_f$- and $P_f List$-coalgebras to the first-order case. In Section 7, we conclude the paper and discuss the significance of the results for *Coinductive Logic Programming* [12,15,16,23,33].

2 Logic Programs

In this section, we recall the essential constructs of logic programming, as described, for instance, in Lloyd's standard text [23]. We start by describing the syntax of logic programs, after which we outline approaches to declarative and operational semantics of logic programs.

Definition 2.1 A *signature* Σ consists of a set of *function symbols* $f, g, \ldots$ each equipped with a fixed *arity*. The arity of a function symbol is a natural number indicating the number of arguments it is supposed to have. Nullary (0-ary) function symbols are allowed: these are called *constants*.

Given a countably infinite set Var of variables, terms are defined as follows.

Definition 2.2 The set $Ter(\Sigma)$ of *terms* over Σ is defined inductively:

- $x \in Ter(\Sigma)$ for every $x \in Var$.
- If f is an n-ary function symbol ($n \geq 0$) and $t_1, \ldots, t_n \in Ter(\Sigma)$, then $f(t_1, \ldots, t_n) \in Ter(\Sigma)$.

Variables will be denoted x, y, z, sometimes with indexes $x_1, x_2, x_3, \ldots$.

Definition 2.3 A *substitution* is a partial function θ from Var to $Ter(\theta)$ with finite domain. Each substitution generates the function

$$\theta(f(t_1, \ldots, t_n)) \equiv f(\theta(t_1), \ldots, \theta(t_n))$$

for every n-ary function symbol f.

We define an *alphabet* to consist of a signature Σ, the set Var, and a set of *predicate symbols* $P, P_1, P_2, \ldots$, each assigned an arity. Let P be a predicate symbol of arity n and $t_1, \ldots, t_n$ be terms. Then $P(t_1, \ldots, t_n)$ is a *formula* (also called an atomic formula or an *atom*). The *first-order language* $\mathcal{L}$ given by an alphabet consists of the set of all formulae constructed from the symbols of the alphabet.

Definition 2.4 Given a first-order language $\mathcal{L}$, a *logic program* consists of a finite set of clauses of the form

$$A \leftarrow A_1, \ldots, A_n,$$

where $A, A_1, \ldots A_n (\ n \geq 0)$ are atoms. The atom A is called the *head* of a clause, and $A_1, \ldots, A_n$ is called its *body*. Clauses with empty bodies are called *unit clauses*.

A *goal* is given by $\leftarrow A_1, \ldots A_n$, where $A_1, \ldots A_n (\ n \geq 0)$ are atoms.

Definition 2.5 A term, an atom, or a clause is called *ground* if it contains no variables. A term t is an *instance* of a term t_1, if there exists a substitution σ such that $\sigma(t_1) = t$; additionally, if t is ground, it is called a *ground instance* of t_1; similarly for atoms and clauses.

Various implementations of logic programs require different restrictions to the first-order signature. One possible restriction is to remove function symbols of arity $n > 0$ from the signature, and the programming language based on such syntax is called DATALOG. The advantages of DATALOG are easier implementations and a greater capacity for parallelisation [35,17,7]. From the point of view of model theory, DATALOG programs always have finite models. Another possible restriction to the syntax of logic programs is *ground logic programs*, that is, there is no restriction to the arity of function symbols, but variables are not allowed. Such restriction yields easier implementations - because there is no need in using unification algorithms, and such programs also can be implemented by using parallel algorithms. We will return to this discussion in Section 4.

Remark 2.6 There are different approaches to the meaning of goals and bodies given by $A, A_1, \ldots A_n$. One approach arises from first-order logic semantics, and treats them as finite conjunctions, in which case the order of atoms is not important, and repetitions of atoms are not considered. Another - practical - approach is to treat bodies as sequences of atoms, in which case repetitions and order can play a role in computations. In Section 4, we will show that both approaches are equally sound in case of ground logic programs, however, the first-order case requires the use of lists; and majority of PROLOG interpreters treat the goals as lists.

We shall now give two examples of logic programs. We shall develop these examples as running examples through the course of the paper.

Example 2.7 Let GC (for graph connectivity) denote the logic program

`connected(x,x)` $\leftarrow$

`connected(x,y)` $\leftarrow$ `edge(x,z)`, `connected(z,y)`.

Here, we used predicates "connected" and "edge" - to make the intended meaning of the program clear.

Additionally, there may be clauses that describe the data base; in our case - edges of a particular graph, e.g.

`edge(a,b)` $\leftarrow$

`edge(b,c)` $\leftarrow$

The latter two clauses are ground, and the atoms in them are ground instances of the atom `edge(x,z)`. A typical goal would be

$$\leftarrow \texttt{connected(a,x)}$$

Example 2.8 Consider the logic program `LN` that inductively describes lists of natural numbers.

$$\begin{aligned}
\texttt{nat(0)} &\leftarrow \\
\texttt{nat(s(n))} &\leftarrow \texttt{nat(n)} \\
\texttt{list(nil)} &\leftarrow \\
\texttt{list(cons(x,y))} &\leftarrow \texttt{nat(x)}, \texttt{list(y)}
\end{aligned}$$

Note that here, inductive definition of `list` depends on the inductive definition of natural numbers.

3 Finite and Infinite Computations by Logic Programs

In this section, we will describe the operational semantics for logic programming given by the algorithm of SLD-resolution. We pay special attention to infinite SLD-derivations, and coinductive logic programs.

Traditionally, logic programming has been given *least fixed point* semantics [23]. Given a logic program P, one lets B_P (also called a *Herbrand base*) denote the set of atomic ground formulae generated by the syntax of P, and one defines the T_P operator on 2^{B_P} by sending I to the set $\{A \in B_P : A \leftarrow A_1, ..., A_n$ is a ground instance of a clause in P with $\{A_1, ..., A_n\} \subseteq I\}$. The least fixed point of T_P is called the *least Herbrand model* of P and duly satisfies model theoretic properties that justify that expression [23].

Note the inefficiency of this. In modelling Example 2.5, we would want to regard `connected(x,x)` as true, but least fixed point semantics does not formally consider even the possibility of truth of such a formula because it is not ground. And the set of all ground instances of an atom in a program can, in general, be infinite. A non-ground alternative to the ground fixed point semantics has been given in [6,5].

The fact that logic programs can be naturally represented via fixed point semantics, has led to the development of *logic programs as inductive definitions*, [27,15,16], as opposed to the view of *logic programs as first-order logic*.

Operational semantics for logic programs is given by SLD-resolution, a goal-oriented proof-search procedure.

Given a substitution θ as in Definition 2.3, and given an atom A, we write $A\theta$ for the atom given by applying the substitution θ to the variables appearing in A. Moreover, given a substitution θ and a list of atoms $(A_1, ...A_k)$, we write $(A_1, ..., A_k)\theta$ for the simultaneous substitution of θ in each atom A_m in the list.

Definition 3.1 Let S be a finite set of atoms. A substitution θ is called a *unifier* for S if, for any pair of atoms A_1 and A_2 in S, applying the substitution θ yields $A_1\theta = A_2\theta$. A unifier θ for S is called a *most general unifier* (mgu) for S if, for each unifier σ of S, there exists a substitution γ such that $\sigma = \theta\gamma$.

Definition 3.2 Let a goal G be $\leftarrow A_1, \ldots, A_m, \ldots, A_k$ and a clause C be $A \leftarrow B_1, \ldots, B_q$. Then G' is *derived* from G and C using mgu θ if the following conditions hold:

- A_m is an atom, called the *selected* atom, in G.
- θ is an *mgu* of A_m and A.
- G' is the goal $\leftarrow (A_1, \ldots, A_{m-1}, B_1, \ldots, B_q, A_{m+1}, \ldots, A_k)\theta$.

Definition 3.3 An *SLD-derivation* of $P \cup \{G\}$ consists of a sequence of goals $G = G_0, G_1, \ldots$, a sequence $C_1, C_2, \ldots$ of instances of program clauses of P and a sequence $\theta_1, \theta_2, \ldots$ of mgus such that each G_{i+1} is derived from G_i and C_{i+1} using θ_{i+1}. An *SLD-refutation* of $P \cup \{G\}$ is a finite SLD-derivation of $P \cup \{G\}$ which has the empty clause $\Box$ as its last goal. If $G_n = \Box$, we say that the refutation has length n.

Operationally, SLD-derivations can be characterised by two kinds of trees — called *SLD-trees* and *proof-trees*, the later called proof-trees for their close relation to proof-trees in e.g. sequent calculus.

Definition 3.4 Let P be a logic program and G be a goal. An *SLD-tree* for $P \cup \{G\}$ is a tree satisfying the following:

(i) Each node of the tree is a (possibly empty) goal.

(ii) The root node is G.

(iii) If $\leftarrow A_1, \ldots, A_m$, $m > 0$ is a node in T; and it has n children, then there exists $A_k \in A_1, \ldots, A_m$ such that A_k is unifiable with exactly n distinct clauses $C_1 = A^1 \leftarrow B^1_1, \ldots, B^1_q$, ..., $C_n = A^n \leftarrow B^n_1, \ldots, B^n_r$ in P via mgus $\theta_1, \ldots \theta_n$, and, for every $i \in \{1, \ldots n\}$, the ith child node is given by the goal

$$\leftarrow (A_1, \ldots, A_{k-1}, B^i_1, \ldots, B^i_q, A_{k+1}, \ldots, A_m)\theta_i$$

(iv) Nodes which are the empty clause have no children.

Each branch of the SLD-tree is a *sequential* derivation of $P \cup \{G\}$. Branches corresponding to successful derivations are called *success branches*, branches corresponding to infinite derivations are called *infinite branches*, branches corresponding to failed derivations are called *failure branches*.

Each SLD-derivation, or, equivalently, each branch of an SLD-tree, can be represented by a proof-tree, defined as follows.

Definition 3.5 Let P be a logic program and $G = \leftarrow A$ be an atomic goal. A *proof-tree* for A is a possibly infinite tree T such that

- Each node in T is an atom.
- A is a root of T.
- For every node A occurring in T, if A has children $C_1, \ldots, C_m$, then there exists a clause $B \leftarrow B_1, \ldots, B_m$ in P such that B and A are unifiable with mgu θ, and $B_1\theta = C_1$, ... ,$B_m\theta = C_m$.

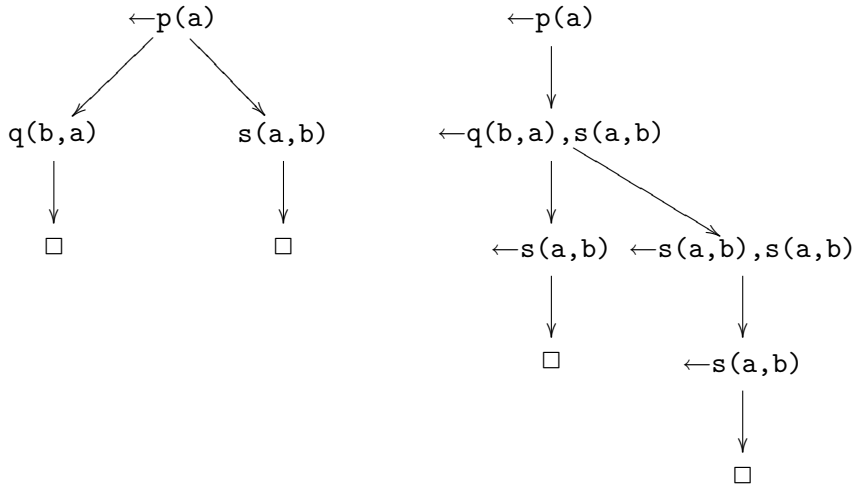
As pointed out in [34], the relationship between proof-trees and SLD-trees is the relationship between deterministic and nondeterministic computations. Whether

the complexity classes defined via proof-trees are equivalent to complexity classes defined via search trees is a reformulation of the classic P=NP problem in terms of logic programming. We will illustrate the two kinds of trees in the following example.

Example 3.6 Consider the following simple ground logic program.

```
q(b,a) ←
s(a,b) ←
  p(a) ← q(b,a), s(a,b)
q(b,a) ← s(a,b)
```

The following diagram shows a proof-tree (on the left) and an SLD-tree (on the right) for this program. The proof-tree corresponds to the left-hand side branch of the SLD-tree.



SLD-resolution is sound and complete with respect to the least fixed point semantics. The classical theorems of soundness and completeness of this operational semantics [23,6,5] show that every atom in the set computed by the least fixed point of T_P has a finite SLD-refutation, and vice versa.

The analysis of the least fixed point operators focuses solely on finite SLD derivations. But infinite SLD derivations are nonetheless common in the practice of programming. There are two kinds of infinite SLD derivations possible: computing finite or infinite objects.

Example 3.7 Consider the logic program from Example 2.7. It is easy to facilitate infinite SLD-derivations, by simply adding a clause that makes the graph cyclic:

$\texttt{edge}(c, a) \leftarrow$

Taking a query $\leftarrow \texttt{connected}(a, z)$ as a goal would lead to an infinite SLD-derivation corresponding to an infinite path starting from a in the cycle. However, the object that is described by this program, the cyclic graph with three nodes, is finite.

Unlike the derivations for the example above, some derivations compute infinite objects.

Example 3.8 The following program `stream` defines the infinite stream of binary bits:

$$\begin{aligned} \texttt{bit}(0) &\leftarrow \\ \texttt{bit}(1) &\leftarrow \\ \texttt{stream(x,y)} &\leftarrow \texttt{bit(x)}, \texttt{stream(y)} \end{aligned}$$

Compare this coinductive program with the inductive program LN from Example 2.8.

The programs like `stream` can be given the declarative semantics via computations of the greatest fixed point of the semantic operator T_P. However the fixed point semantics is incomplete in general [23]: it fails for some infinite derivations.

Example 3.9 The program below will be characterised by the greatest fixed point of the T_P operator that contains $R(f^\omega(a))$, whereas no infinite term will be computed via SLD-resolution.

$$R(x) \leftarrow R(f(x))$$

There have been numerous attempts to resolve the mismatch between infinite derivations and greatest fixed point semantics, [36,12,15,16,23,27,33]. But, the infinite SLD derivations of both finite and infinite objects have not yet received a uniform semantics, see Figure 1.

In [19,20] we described algebraic fibrational semantics and proved soundness and completeness result for it with respect to finite SLD-refutations, see Figure 1. In this paper, we adapt the algebraic fibrational semantics to give coalgebraic treatment of both finite and infinite SLD derivations.

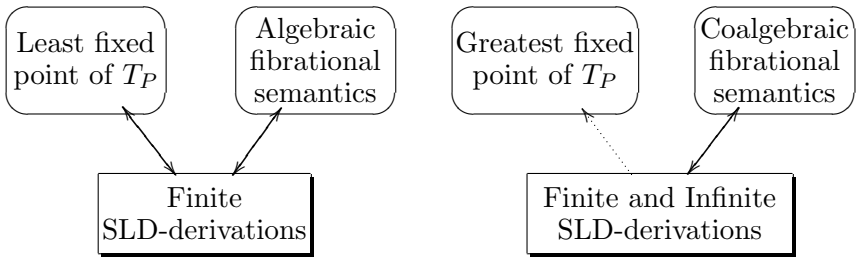


Fig. 1. Alternative declarative semantics for finite and infinite SLD-derivations. The solid arrows show the semantics that both sound and complete, and the dotted arrow indicates sound incomplete semantics.

4 Coalgebraic Semantics for Ground Logic Programs

In this section, we consider a coalgebraic semantics for logic programs. This semantics is intended to give meaning to both finite and infinite SLD-derivations. We restrict our attention to SLD-derivations for ground logic programs, and defer analysis of variables, substitution and unification until the next section.

Example 4.1 A ground logic program equivalent to the program from Example 2.7 can be obtained by taking all ground instances of non-ground clauses, such as, for example

$$\begin{aligned}
&\texttt{connected}(a,a) \leftarrow \\
&\texttt{connected}(b,b) \leftarrow \\
&\qquad\qquad\vdots \\
&\texttt{connected}(a,b) \leftarrow \texttt{edge}(b,c), \texttt{connected}(c,b) \\
&\texttt{connected}(a,c) \leftarrow \texttt{edge}(c,b), \texttt{connected}(b,c) \\
&\qquad\qquad\vdots
\end{aligned}$$

When a non-ground program contains no function symbols (cf. Example 2.7), there always exists a finite ground program equivalent to it, such as the one in the example above. In this section, we will work only with finite ground logic programs. Finite ground logic programs can still give rise to infinite SLD-derivations. If the graph described by a logic program above is cyclic, it can give rise to infinite SLD-derivations, see Example 3.7.

Variable-free logic programs and SLD derivations over such programs bear strong resemblance to finitely branching transition systems: atoms play the role of states, and the implication arrow $\leftarrow$ used to define clauses plays the role of a transition relation. Note that the SLD-resolution is the goal-oriented procedure, which, given a goal $G_1 = A$, uses a clause $A \leftarrow A_1, \ldots, A_n$ to form a new goal $G_2 = A_1, \ldots, A_n$, and continues recursively. The main difference between logic programs and transition systems is that each atom is sent to the subset of subsets of atoms appearing in the program, and thus the powerset functor is iterated twice.

Construction 4.1 *Given a variable-free logic program P, let At be the set of all atoms appearing in P. Then P can be identified with a P_fP_f-coalgebra (At, p), where $p : At \longrightarrow P_f(P_f(At))$ sends an atom A to the set of bodies of those clauses in P with head A.*

Remark 4.2 Note that by Remark 2.6, one can alternatively view the bodies of clauses as lists of atoms. In this case, Construction 4.1 can alternatively be given by P_fList-coalgebra. Namely, a ground logic program P can be identified with a P_fList-coalgebra (At, p), where $p : At \longrightarrow P_f(List(\mathrm{At}))$ sends an atom A to the set of bodies of those clauses in P with head A.

One can use the cofree comonad $C(P_fP_f)$ on P_fP_f and the $C(P_fP_f)$-coalgebra determined by the P_fP_f-coalgebra to give an account of derivation trees for the atoms appearing in a given logic program P. This bears comparison with the greatest fixed point semantics for logic programs, see e.g. [23], but we do not pursue that here. The following theorem uses the construction given in [18,37].

Theorem 4.3 *Given an endofunctor $H : Set \longrightarrow Set$ with a rank, the forgetful functor $U : H - Coalg \longrightarrow Set$ has a right adjoint R.*

Construction 4.2 *R is constructed as follows. For a given $Y \in Set$, put $Y_0 = Y$,*

and $Y_{\alpha+1} = Y \times H(Y_\alpha)$. *We define* $\delta_\alpha : Y_{n+1} \longrightarrow Y_n$ *inductively by*

$$Y_{\alpha+1} = Y \times HY_\alpha \overset{Y \times H\delta_{\alpha-1}}{\longrightarrow} Y \times HY_{\alpha-1} = Y_\alpha,$$

with the case of $\alpha = 0$ *given by the map* $Y_1 = Y \times HY \overset{\pi_1}{\longrightarrow} Y$. *For a limit ordinal, let* $Y_\alpha = \lim_{\beta<\alpha}(Y_\beta)$, *determined by the sequence*

$$Y_{\beta+1} \overset{\delta_\beta}{\longrightarrow} Y_\beta.$$

If H *has a rank, there exists* α *such that* Y_α *is isomorphic to* HY_α *and forms the cofree coalgebra on* Y.

Corollary 4.4 *If* H *has a rank,* U *has a right adjoint* R *and putting* $G = RU$, G *possesses a canonical comonad structure and there is a coherent isomorphism of categories*

$$G - Coalg \cong H - Coalg,$$

where $G - Coalg$ *is the category of* G*-coalgebra for the comonad* G.

We can apply the general results and construction given by Theorem 4.3 and Corollary 4.4 to our analysis of logic programs and $P_f P_f$-coalgebras. Taking $p : \mathrm{At} \longrightarrow P_f P_f(\mathrm{At})$, by the construction of Theorem 4.3, the corresponding G-coalgebra where G is the cofree comonad on $P_f P_f$ is given as follows: $G(\mathrm{At})$ is given by a limit of the form

$$\ldots \longrightarrow \mathrm{At} \times P_f P_f(\mathrm{At} \times P_f P_f(\mathrm{At} \times P_f P_f(\mathrm{At}))) \longrightarrow \mathrm{At} \times P_f P_f(\mathrm{At}) \longrightarrow \mathrm{At}.$$

And $\overline{p} : \mathrm{At} \longrightarrow G(\mathrm{At})$ sends an element A of At to an element of this limit.

Note that by Remarks 2.6 and 4.2, a similar construction of G-coalgebra but for $P_f List$-coalgebra instead of $P_f P_f$-coalgebra can be used to give semantics to SLD derivations for logic programs whose bodies are treated as lists of atoms.

Construction 4.3 *Both approaches can be described by putting*

- $p_0(A) = A$,
- $p_1(A) = (A, \{$*bodies of clauses with the head* $A\})$; *and*
- $p_2(A) = (A, \{$*bodies of clauses with the head* A, *together with, for each formula* A_{ij} *in the body of each clause with the head* A, $\{$ *bodies of clauses with the head* $A_{ij}\}\})$;
- *etc.*

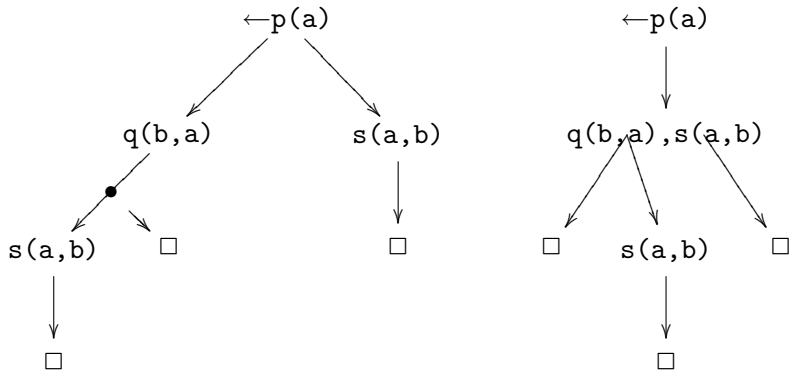
Construction 4.3 describes a structure that resembles derivation trees used in logic programming, but as we show in the next example, these are not the traditional proof-trees or SLD-trees from Definitions 3.5 and 3.4.

Example 4.5 Consider the logic program from Example 3.6.

The program has three atoms, namely `q(b,a)`, `s(a,b)` and `p(a)`. So $At = \{$`q(b,a)`, `s(a,b)`, `p(a)`$\}$. And the program can be identified with the $P_f P_f$-coalgebra structure on At given by
$p($`q(b,a)`$) = \{\{\}, \{$`s(a,b)`$\}\}$, where $\{\}$ is the empty set.

$p(\texttt{s(a,b)}) = \{\{\}\}$, i.e., the one element set consisting of the empty set.
$p(\texttt{p(a)}) = \{\{\texttt{q(b,a)},\texttt{s(a,b)}\}\}$.

Consider the $C(P_f P_f)$-coalgebra corresponding to p. It sends `p(a)` to a parallel refutation of `p(a)` as depicted in the left-hand diagram below. Note that in the diagram, we use traditional notation $\Box$ to denote $\{\}$ in the derivation trees.



In contrast, considering the program as a $P_f List$-coalgebra, we would have:
$p(\texttt{q(b,a)}) = \{[nil], [\texttt{s(a,b)}]\}$, i.e., the set of two lists, with nil being the empty list.
$p(\texttt{s(a,b)}) = \{[nil]\}$.
$p(\texttt{p(a)}) = \{[\texttt{q(b,a) :: s(a,b)}]\}$, i.e, the set containing one list $[\texttt{q(b,a) :: s(a,b)}]$; with the corresponding $C(P_f List)$-coalgebra sending `p(a)` to the sequential refutation of `p(a)` as depicted in the right-hand side diagram above. Note that in the diagram, traditional notation $\Box$ is used for nil; and `,` is used for `::`.

Note that the derivation trees shown in Example 4.5 differ from the SLD-tree (cf. Definition 3.4) and the proof-tree (cf. Definition 3.5) constructed for the same logic program in Example 3.6. The reason is that the derivations modelled above by the G-coalgebra have strong relation to *parallel logic programming*, [35,17], while both proof-trees and SLD-trees describe sequential derivation strategies.

One of the distinguishing features of logic programming languages is that they allow implicit parallel execution of programs. The two main types of parallelism used in implementations of logic programs are *and-parallelism* and *or-parallelism*, see [11,10,30] for an excellent analysis of the two kinds of parallelism in logic programming.

Or-parallelism arises when more than one clause unifies with the goal atom — the corresponding bodies can be executed in Or-parallel fashion. Or-parallelism is thus a way of efficiently searching for solutions to a goal, by exploring alternative solutions in parallel. It corresponds to the parallel execution of the branches of an SLD-tree, cf Definition 3.4 and Example 3.6. Or-parallelism has successfully been exploited in Prolog in the Aurora [24] and the Muse [1] systems both of which have shown very good speed-up results over a considerable range of applications.

(Independent) And-parallelism arises when more than one atom is present in the goal, and the atoms do not share variables. In this section, we consider only ground logic programs, and so the latter condition is satisfied trivially. That is, given a

goal $G = \leftarrow B_1, \ldots B_n$, an *And-parallel algorithm* of SLD resolution looks for SLD derivations for each of B_i simultaneously. Independent And-parallelism is thus a way of splitting up a goal into subgoals, and corresponds to the parallel computation of all branches in a proof-tree, see Definition 3.5 and Example 3.6. Independent And-parallelism has been successfully exploited in the &-Prolog System [14].

The comonad we have constructed in this section models a synthetic form of parallelism - *And-Or parallelism*. The most common way to express And-Or parallelism in logic programs is through traditional *and-or trees* [10], which consist of *or-nodes* and *and-nodes*. Or-nodes represent multiple clause heads unifying with a goal atom; while and-nodes represent multiple subgoals in the body of a clause being executed in and-parallel. And-Or parallel prolog was first implemented as Andorra system [3], with many more implementations following it [10].

Definition 4.6 Let P be a logic program and $G = \leftarrow A$ be an atomic goal. A *parallel and-or tree* for A is a possibly infinite tree T satisfying the following properties.

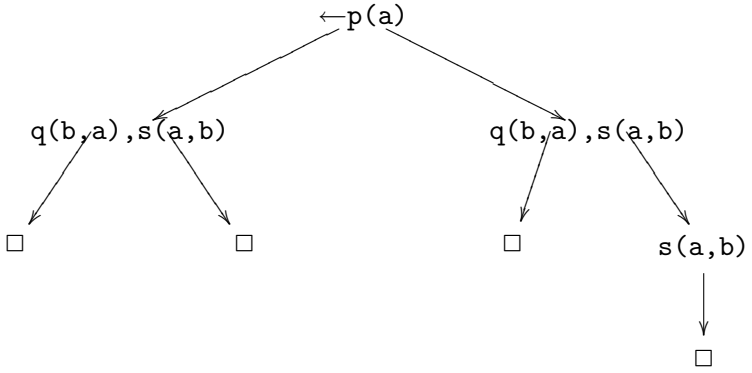
- A is a root of T.
- Each node in T is either an and-node or an or-node.
- Each or-node is given by $\bullet$.
- Each and-node is an atom.
- For every node A occurring in T, if A is unifiable with only one clause $B \leftarrow B_1, \ldots, B_n$ in P with mgu θ, then A has n children given by and-nodes $B_1\theta, \ldots B_n\theta$.
- For every node A occurring in T, if A is unifiable with exactly $m > 1$ distinct clauses $C_1, \ldots, C_m$ in P via mgus $\theta_1, \ldots, \theta_m$, then A has exactly m children given by or-nodes, such that, for every $i \in m$, if $C_i = B^i \leftarrow B^i_1, \ldots, B^i_n$, then the ith or-node has n children given by and-nodes $B^i_1\theta_i, \ldots, B^i_n\theta_i$.

Depending on whether the comonad is constructed for $P_f P_f$ or $P_f List$-coalgebra, it models different kinds of and-or parallel derivation trees. Using $P_f P_f$-coalgebra, we model traditional and-or trees, as given in Definition 4.6 and illustrated in Example 4.5, in the left-hand side derivation tree.

The action of the comonad G on $P_f List$ bears some resemblance with the *Composition (and-or parallel) Trees*. The Composition trees were introduced in [10] as a way to simplify the implementation of the traditional and-or trees which required a lot of machinery to synchronise branches that were created by or-nodes. The composition trees contain *composition* nodes used whenever both and- and or-parallel computations are possible for one goal. Every composition node is a list of atoms in the goal. If, in a goal $G = \leftarrow B_1, \ldots B_n$, an atom B_i is unifiable with $k > 1$ clauses, then one adds k children (k composition nodes) to the node G; similarly for every atom in G that is unifiable with more than one clause. Every such composition node has the form $B_1, \ldots B_n$, and n and-parallel edges. Thus, all possible combinations of all possible or-choices at every and-parallel step are given.

Example 4.7 Continuing Examples 3.6 and 4.5, the composition tree for the pro-

gram will look as follows.



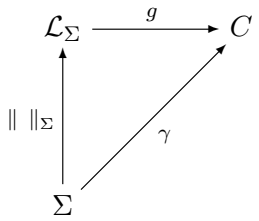
In this Section, we have shown that the construction of G-coalgebra on $P_f P_f$ and $P_f List$ can model different types of parallel and-or derivation trees. Logic programs give the user a variety of ways in which the execution of ground logic programs can be parallelised; and the coalgebraic constructions we have proposed can be adapted to model different kinds of parallel derivation trees.

5 First-Order Language and Lawvere Theory

In this section, we consider the first-order signature Σ and Lawvere theory generated by it; cf. [19].

Definition 5.1 Given a signature Σ as in Definition 2.1 and given a category C with strictly associative finite products, an *interpretation* of Σ in C is an object X of C, together with, for each function symbol f of arity n, a map in C from X^n to X.

Proposition 5.2 *Given Σ, there exists a category $\mathcal{L}_\Sigma$ with strictly associative finite products and an interpretation $\| \ \|_\Sigma$ of Σ in $\mathcal{L}_\Sigma$, such that for any category C with strictly associative finite products, and interpretation γ of Σ in C, there exists a unique functor $g : \mathcal{L}_\Sigma \to C$ that strictly preserves finite products, such that g composed with $\| \ \|_\Sigma$ gives γ, as in the following diagram:*



Proof.

Define the set $\mathtt{ob}(\mathcal{L}_\Sigma)$ to be the set of natural numbers.

To each natural number n, let $x_1, \ldots, x_n$ be a specified list of distinct variables. Then define $\mathtt{ob}(\mathcal{L}_\Sigma)(n, m)$ to be the set of m-tuples $(t_1, \ldots, t_m)$ of terms generated by the function symbols in Σ and variables $x_1, \ldots, x_n$. Define composition in $\mathcal{L}_\Sigma$ by substitution.

One readily check that this satisfies the axioms for a category and $\mathcal{L}_\Sigma$ has strictly associative finite products, given by the sum of natural numbers. Its universal property follows directly from its construction.

□

One can describe $\mathcal{L}_\Sigma$ without the need for a specified list of variables for each n: one can instead posit a countable set of variables as usual, consider all terms in n variables, and factor out by α-equivalence. Observe that the terminal object of $\mathcal{L}_\Sigma$ is the natural number 0.

The category $\mathcal{L}_\Sigma$ is often called the *Lawvere theory* generated by Σ; [21].

Example 5.3 Consider the logic program GC from Example 2.7, Then the objects of $\mathcal{L}_\Sigma$ are natural numbers, the constants a, b, and c are modelled by elements of $\mathtt{ob}(\mathcal{L}_\Sigma)(0,1)$; additional arrows are given by projections $\pi_i : n \to 1$.

Example 5.4 The program from Example 3.8 will have a similar semantics: there, `0` and `1` play a role of a and b in GC; and, as well as GC, the program contains only nullary function symbols.

For simplicity of exposition, we have only dealt with the untyped situation here, but one can generalise $\mathcal{L}_\Sigma$ to a small category $\mathcal{C}_\Sigma$ with finite products if the signature is sorted, see [20].

Example 5.5 Consider the logic program `LN` from Example 2.8. This is naturally two-sorted, with one sort for natural numbers and one sort for lists. So traditionally, category theory would not use Proposition 4.2 but rather a multi-sorted version of it; see [20]. But Example 2.8 is a legitimate untyped logic program and is representative of such. And, as such, it is an example of the framework we propose, just not organised in the way traditionally done by category theorists.

The constants `0` and `nil` are duly modelled by maps from 0 to 1 in $\mathcal{L}_\Sigma$, `s` is modelled by a map from 1 to 1, and `cons` is modelled by a map from 2 to 1. The term `s(0)` is therefore modelled by a map from 0 to 1, as is the term `s(nil)`, although the latter does not make semantic sense; the alpha-equivalence class of `cons(x,nil)` is modelled by a map from 1 to 1; and `cons(0,nil)` is modelled by a map from 0 to 1, as is `cons(nil,nil)`, although it again does not make semantic sense, all of this determined by composition in $\mathcal{L}_\Sigma$. The sorting is modelled *not* by having a sorted finite product theory but rather by predicates, which are modelled in the category $Set^{\mathcal{L}_\Sigma}$, or, more subtly, in $Poset^{\mathcal{L}_\Sigma}$.

Note that $\mathcal{L}_\Sigma$ is the *Lawvere theory* generated by Σ. The structure of $\mathcal{L}_\Sigma$ allows us to characterise most general unifiers from definition 3.1 in terms of equalisers, as follows, see also [32].

Proposition 5.6 *Given a signature Σ, if $u : n \to 1$ and $u' : n \to 1$ are arrows in $\mathcal{L}_\Sigma$, a pair (p, v), where p is an object of $\mathcal{L}_\Sigma$ and v is an arrow $v : p \to n$, is a unifier of u and u' if $u \circ v = u' \circ v$. Furthermore, (p, v) is a most general unifier of u and u' if, for any unifier (q, w) of u and u', there exists a unique arrow $f : q \to p$ satisfying $v \circ f = w$.*

So, defined semantically, mgu's are exactly equalisers in $\mathcal{L}_\Sigma$. Note that in [32],

most general unifiers are modelled by coequalisers in the Kleisli category for a monad T_Σ on Set. The monad T_Σ corresponds to the Lawvere theory $\mathcal{L}_\Sigma$: taking the opposite of the Kleisli category $K(T_\Sigma)$, and restricting it to natural numbers would yield $\mathcal{L}_\Sigma$.

Example 5.7 The most general unifier for the terms `cons(x,nil)` and `cons(s(0),y)` from Example 2.8, is the substitution $\sigma : \{\texttt{x/s(0)}, \texttt{y/nil}\}$. As explained in Example 5.5, *a priori*, the alpha-equivalence class of `cons(x,nil)` is modelled by a map from 1 to 1, similarly for that of `cons(s(0),y)`. But we do *not* model the mgu of the two terms simply by attempting to take an equaliser (which does not exist!) of those two maps qua maps from 1 to 1 in $\mathcal{L}_\Sigma$.

What we do is more subtle than that. We first put `cons(x,nil)` and `cons(s(0),y)` in context, the context being the presence of two variables `x` and `y`. So we consider the following two maps: one given by the composite of $\pi_1 : 2 \longrightarrow 1$ with the α-equivalence class of `cons(x,nil)` $: 1 \longrightarrow 1$, the other given by the composite of $\pi_2 : 2 \longrightarrow 1$ with the alpha-equivalence class of `cons(s(0),y)` $: 1 \longrightarrow 1$, then we take the equaliser of those two maps from 2 to 1. That equaliser does exist and is given by the pair `(s(0),nil)`.

This definition of a most general unifier provides semantic support for the usual algorithm for computing most general unifiers, [23].

6 Coalgebraic Semantics for First-Order Logic Programs

In this Section, we extend the coalgebraic semantics for ground logic programs given by $P_f P_f$- and G-coalgebra in Section 4 to the general first-order case. We will build upon the Lawvere theory $\mathcal{L}_\Sigma$ we defined for the first-order signature Σ in Section 5.

When one models substitution, difficulty arises in the finiteness associated with the functors $P_f P_f$ and $P_f List$ as substitution can generate infinitely many instances of clauses with the same head. For instance, suppose one had a clause of the form

$$A \leftarrow R(x).$$

For any constant c and any unary function symbol f, substitution yields the clause

$$A \leftarrow R(f^n(c))$$

for every natural number n, thus a countable set of clauses with head A. The reason for introducing Lawvere theories is precisely to allow for substitution. So we do need to allow for possibilities such as this, as the infiniteness arises even from a finite alphabet.

So, in extending from the ground to the non-ground case, we cannot simply apply $P_f P_f$ or $P_f List$ pointwise, but we must extend them to $P_c P_f$ and $P_c List$ respectively.

A further problem arises in that, for instance using the alphabet of the above example, the clause

$$A \leftarrow R(c)$$

might be in the logic program, while the clause

$$A \leftarrow R(x)$$

might not be in the logic program and cannot be deduced from it. Examples like this mean that we cannot simply model a logic program by a coalgebra for the functor on $Set^{\mathcal{L}_\Sigma}$ sending an object X to $P_c P_f\ X$: the coalgebra map would be a natural transformation, and the naturality would disallow examples like the above. This difficulty is more complex than the previous one, and it seems best resolved by the use of "laxness", which in turn requires us to generalise from Set to $Poset$.

Definition 6.1 Given locally ordered functors $H, K : C \longrightarrow D$, a *lax natural transformation* from H to K is the assignment to each object c of C, of a map $\alpha_c : Hc \longrightarrow Kc$ such that for each map $f : c \longrightarrow c'$ in C, one has $(Kf)(\alpha_c) \leq (\alpha'_c)(Hf)$.

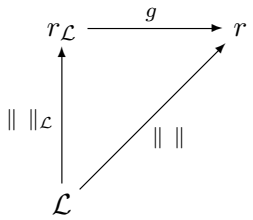
Definition 6.2 An *indexed poset* over a small category C is a locally ordered functor $r : C \to \mathrm{Poset}$. An *indexed map of posets* from r to q is a lax natural transformation $\tau : r \Rightarrow q : C \to Poset$. An indexed map of posets τ is *strict* if it is a natural transformation (not just lax).

Indexed posets and lax natural transformations, with pointwise composition and pointwise ordering, form a locally ordered category we denote by $Lax(C, Poset)$. We model non-ground logic programs by using coalgebra on the underlying category of $Lax(C, Poset)$ relative to a functor given by post-composing with the functor $P_c P_f$ suitably extended to $Poset$.

We start by giving an account to the first-order atoms. Note that the ground case we described in Section 4 was equal to the propositional case, and the set of atomic formulae At was treated as a set of propositions; for example, $R(a)$ and $R(b)$ were simply different elements of At. The full fragment of first-order language requires a more careful analysis of predicates and terms appearing in them: $R(x)$, $R(f(x))$, and $R(a)$ are all dependent atomic formulae, and their dependencies, as well as their place in clauses, play role in computations, see Example 6.6. Therefore, we start by giving an account to first-order language prior to introducing clauses.

Definition 6.3 An *interpretation of a language* $\mathcal{L}$ in $r : \mathcal{L}_\Sigma^{op} \to Poset$, is given by the interpretation $\| \ \|_\Sigma$: of Σ in $\mathcal{L}_\Sigma$ (cf. Proposition 5.2), together with, for each predicate R of arity n, an element $\|R\|$ of $ob(r(n))$.

Proposition 6.4 *Given a language $\mathcal{L}$, there exists an indexed poset $r_\mathcal{L}$, and an interpretation $\| \ \|_\mathcal{L}$ of $\mathcal{L}$ in $r_\mathcal{L}$, such that the following universal property holds. For any indexed poset r over $\mathcal{L}_\Sigma$ and any interpretation $\| \ \|$ of $\mathcal{L}$ in r, there exists a unique strict map of posets $g : r_\mathcal{L} \to r$ such that g composed with $\| \ \|_\mathcal{L}$ gives $\| \ \|$, as on the following diagram:*



Proof. For each $n \epsilon \mathcal{L}_\Sigma$, $r_\mathcal{L}(n)$ is the set of atomic formulae in $\mathcal{L}$ with variables taken from a specified list $(x_1, \ldots, x_n)$, with the trivial ordering, with functoriality given by substitution. The universal property holds as each such atomic formula is uniquely determined by the alphabet of the language, and naturality forces substitution to be respected.

□

The construction can alternatively be expressed in terms of factoring out by α-equivalence as in Proposition 5.2. In modelling an arbitrary logic program written in the language $\mathcal{L}$, the indexed poset $r_\mathcal{L}$ plays the role for us that the set At of atomic formulae played when we restricted our attention to variable-free logic programs in Section 4.

Note that the construction of $r_\mathcal{L}$ corresponds to what is known as *non-ground Herbrand base*, and was proposed in [6,5] as an alternative to the semantics based upon traditional, i.e., ground Herbrand base. Moreover, it has been proven in [5,6] that the fixed point semantics given by means of the non-ground Herbrand base yields soundness and completeness theorems for the SLD-resolution.

Example 6.5 For the logic program GC from Examples 2.7 and 5.3, $r_\mathcal{L}$ is as follows:

- $r_\mathcal{L}(0)$ is the set of ground atomic formulae built from the underlying language of GC. So its elements are all of the form `edge(a,b)` or `connected(a,b)` for any constants `a` and `b`; note that `a` may equal `b`.
- $r_\mathcal{L}(1)$ is the set of atomic formulae with at most one free variable, built from the underlying language of GC using a single specified variable `x`. So it includes all ground atomic formula `edge(a,b)`, `connected(a,b)`, but also includes, up to α-equivalence, all formulae of the form `edge(a,x)`, `edge(x,a)`, `edge(x,x)`, `connected(a,x)`, `connected(x,a)` and `connected(x,x)`.
- $r_\mathcal{L}(2)$ consists of all of the above, but also `edge(x,y)`, `edge(y,x)`, `connected(x,y)`, `connected(y,x)`, and all non-trivial instances of the above with `x` replaced by `y`.
- $r_\mathcal{L}(3)$ is essentially the same as $r_\mathcal{L}(2)$ except that one now has a third specified variable `z` and one must systematically allow it to play the same role as `x` or `y`, but as a third place-holder rather than the first or second.
- etcetera.

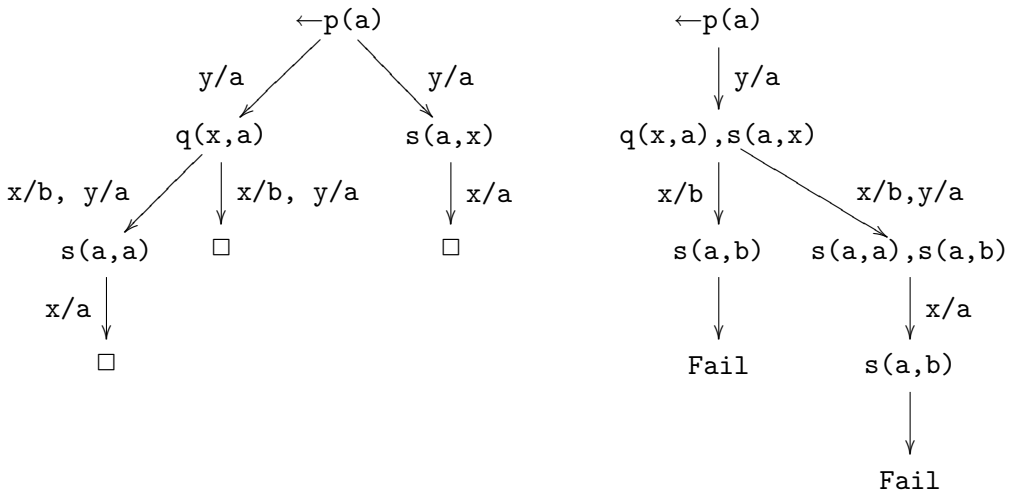
We believe that it is possible to reformulate Construction 4.1 using the indexed poset $r_\mathcal{L}$ to model the set of atoms At, and we plan to give the formal account of this construction in the future.

If we wish to extend the construction of G-coalgebra of Section 4 to the first-order case, using the $P_c P_f$-coalgebra, we have to bear in mind that G-coalgebra gave an account to and-or parallel derivation trees for ground logic programs. Arbitrary first-order logic programs are not as easily parallelisable: the presence of function symbols, variable dependencies, and the use of unification make logic programming *P-complete* and hence inherently sequential, [4,17].

Example 6.6 Consider the following non-ground logic program which is a modification of the ground program from Example 4.5.

```
q(b,y) ←
s(x,x) ←
  p(y) ← q(b,y), s(y,x)
q(b,y) ← s(y,y)
```

Below are the parallel And-Or derivation and a sequential refutation for the query ← p(a):



As one can see, the parallel computation succeeds, and the sequential computation fails.

The example above illustrates that in presence of variables, parallel computations are not guaranteed to be sound. In practice, a mixture of And-parallel, Or-Parallel, and sequential derivations steps are used when non-ground logic programs are processed in a parallel way, see [10].

7 Conclusions

In this paper, we have modelled the operational semantics of variants of PROLOG by coalgebra, with the bulk of our work devoted to modelling variable-free programs. We plan to fully extend the coalgebraic analysis of ground logic programs to non-ground logic programs. In Section 6, we have already given some explanation of how this can be done. The formal coalgebraic analysis of the full fragment of first-order logic programs is a subject for another paper.

The key fact driving our analysis has been the observation that the implication ← acts at a meta-level, like a sequent rather than a logical connective. That observation extends to first-order fragments of linear logic and the Logic of Bunched Implications [8,31]. So we plan to extend the work in the paper to logic programming languages based on such logics.

The situation regarding higher-order logic programming languages such as λ-*PROLOG* [25] is more subtle. Despite their higher-order nature, such logic pro-

gramming languages typically make fundamental use of sequents. So it may well be fruitful to consider modelling them in terms of coalgebra too, albeit probably on a sophisticated base category such as a category of Heyting algebras.

In a separate direction, we have asserted that we have modelled operational semantics of logic programming. The assertion was in the context of the usual use of the term operational semantics in the logic programming literature. But we have not given a formal operational semantics along the lines of rules for *Structural Operational Semantics* [13,28,29] and stated and proved a precisely formulated theorem relating such formal operational semantics with coalgebra. So we plan to develop the combination of Definition 3.2 and Constructions 4.1 and 4.3 to produce such a formal account. We would then be able to compare the formal results with other uses of coalgebra to model operational semantics, e.g., [22].

Another area of research would be to investigate the operational meaning of the coinductive logic programming [2,9,33] that requires a slight modification to the algorithm of the SLD-resolution we have considered in this paper. In particular, the interpreter for coinductive logic programs of this kind would be able to deduce a finite atom `stream(x,y)` from the infinite derivation of a stream defined in Example 3.8.

Another issue we intend to investigate is possible proofs for the theorems of soundness and completeness of the infinite SLD-derivations relative to the coalgebraic semantics we have proposed here. As we explained in Section 2, and illustrated in Example 3.9 and Figure 1, the infinite SLD-derivations are incomplete with respect to the traditional greatest fixed point semantics, because atoms of the form $R(f^{\omega}(a))$ can be computed at the greatest fixed point of the semantic operator, but they cannot be computed by the SLD resolution. The coalgebraic semantics we have proposed here may suit better to give the meaning to the operational semantics of both inductive and coinductive logic programming: in particular $\mathcal{L}_{\Sigma}$ does not model infinite terms.

References

[1] K. Ali and R. Karlsson. Full prolog and scheduling or-parallelism in muse. *Int. Journal Of Parallel Programming*, 19(6):445–475, 1991.

[2] D. Ancona, G. Lagorio, and E. Zucca. Type inference by coinductive logic programming. In S. Berardi, F. Damiani, and U. de'Liguoro, editors, *Types for Proofs and Programs, International Conference, TYPES 2008, Torino, Italy, March 26-29, 2008, Revised Selected Papers*, volume 5497 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2009.

[3] V. S. Costa, D. H. D. Warren, and R. Yang. Andorra-I: A parallel prolog system that transparently exploits both and- and or-parallelism. In *PPOPP*, pages 83–93, 1991.

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

[5] M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. A model-theoretic reconstruction of the operational semantics of logic programs. *Inf. Comput.*, 103(1):86–113, 1993.

[6] M. Falaschi, G. Levi, C. Palamidessi, and M. Martelli. Declarative modeling of the operational behavior of logic languages. *Theor. Comput. Sci.*, 69(3):289–318, 1989.

[7] H. Garcia-Moulina, J. Ullman, and J. Widom. *Database sysyems. The complete book.* Prentice Hall, 2002.

[8] J.-Y. Girard. Linear logic. *Theor. Comput. Sci.*, 50:1–102, 1987.

[9] G. Gupta, A. Bansal, R. Min, L. Simon, and A. Mallya. Coinductive logic programming and its applications. In *Logic Programming, 23rd International Conference, ICLP 2007, Porto, Portugal, September 8-13, 2007, Proceedings*, volume 4670 of *Lecture Notes in Computer Science*, pages 27–44. Springer, 2007.

[10] G. Gupta and V. S. Costa. Optimal implementation of and-or parallel prolog. In *Conference proceedings on PARLE'92*, pages 71–92, New York, NY, USA, 1994. Elsevier North-Holland, Inc.

[11] G. Gupta and B. Jayaraman. Analysis of or-parallel execution models. *ACM Trans. Program. Lang. Syst.*, 15(4):659–680, 1993.

[12] J. Hein. Completions of perpetual logic programs. *Theor. Comput. Sci.*, 99(1):65–78, 1992.

[13] M. Hennessy. *The Semantics of Programming Languages: an Elementary Introduction using Structural Operational Semantics*. John Wiley and Sons, New York, N.Y., 1990.

[14] M. V. Hermenegildo and K. J. Greene. &-prolog and its performance: Exploiting independent and-parallelism. In *ICLP*, pages 253–268, 1990.

[15] M. Jaume. Logic programming and co-inductive definitions. In *CSL*, pages 343–355, 2000.

[16] M. Jaume. On greatest fixpoint semantics of logic programming. *J. Log. Comput.*, 12(2):321–342, 2002.

[17] P. C. Kanellakis. Logic programming and parallel complexity. In *Foundations of Deductive Databases and Logic Programming.*, pages 547–585. Morgan Kaufmann, 1988.

[18] G. M. Kelly. A unified treatment of transfinite constructions for free algebras, free monoids, colimits, associated sheaves, and so on. *Bull. Austral. Math. Soc.*, 22:1–83, 1980.

[19] Y. Kinoshita and A. J. Power. A fibrational semantics for logic programs. In R. Dyckhoff, H. Herre, and P. Schroeder-Heister, editors, *Proceedings of the Fifth International Workshop on Extensions of Logic Programming*, volume 1050 of *LNAI*, Leipzig, Germany, 1996. Springer.

[20] E. Komendantskaya and J. Power. Fibrational semantics for many-valued logic programs: Grounds for non-groundness. In *JELIA*, pages 258–271, 2008.

[21] W. Lawvere. *Functional semantics of algebraic theories*. PhD thesis, Columbia University, 1963.

[22] M. Lenisa, J. Power, and H. Watanabe. Category theory for operational semantics. *Theor. Comput. Sci.*, 327(1-2):135–154, 2004.

[23] J. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 2nd edition, 1987.

[24] E. L. Lusk, D. H. D. Warren, and S. Haridi. The aurora or-parallel prolog system. *New Generation Computing*, 7(2,3):243–273, 1990.

[25] D. Miller and G. Nadathur. Higher-order logic programming. In *ICLP*, pages 448–462, 1986.

[26] G. Nadathur and D. Miller. Higher-order Horn clauses. *Journal of the Association for Computing Machinery*, 37(4):777–814, October 1990.

[27] L. C. Paulson and A. W. Smith. Logic programming, functional programming, and inductive definitions. In *ELP*, pages 283–309, 1989.

[28] G. D. Plotkin. The origins of structural operational semantics. *J. Log. Algebr. Program.*, 60-61:3–15, 2004.

[29] G. D. Plotkin. A structural approach to operational semantics. *J. Log. Algebr. Program.*, 60-61:17–139, 2004.

[30] E. Pontelli and G. Gupta. On the duality between or-parallelism and and-parallelism in logic programming. In *Euro-Par*, pages 43–54, 1995.

[31] D. Pym. *The Semantics and Proof Theory of the Logic of Bunched Implications*, volume 26 of *Applied Logic Series*. Kluwer Academic Publishers, 2002.

[32] D. Rydeheard and R. Burstall. *Computational Category theory*. Prentice Hall, 1988.

[33] L. Simon, A. Bansal, A. Mallya, and G. Gupta. Co-logic programming: Extending logic programming with coinduction. In *ICALP*, volume 4596 of *LNCS*, pages 472–483. Springer, 2007.

[34] L. Sterling and E. Shapiro. *The art of Prolog*. MIT Press, 1986.

[35] J. D. Ullman and A. V. Gelder. Parallel complexity of logical query programs. *Algorithmica*, 3:5–42, 1988.

[36] M. H. van Emden and M. A. N. Abdallah. Top-down semantics of fair computations of logic programs. *J. Log. Program.*, 2(1):67–75, 1985.

[37] J. Worrell. Toposes of coalgebras and hidden algebras. *Electr. Notes Theor. Comput. Sci.*, 11, 1998.