Coalgebraic Semantics for Parallel Derivation Strategies in Logic Programming

Ekaterina Komendantskaya[1], Guy McCusker[2], and John Power[2]

[1] Department of Computing, University of Dundee, UK *
[2] Department of Computer Science, University of Bath, UK **

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 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.
Key words: Logic programming, SLD-resolution, Parallel Logic programming, Coalgebra, Coinduction.

1 Introduction

In the standard formulations of logic programming, such as in Lloyd's book [19], a first-order 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 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.

* The work was supported by the Engineering and Physical Sciences Research Council, UK; Postdoctoral Fellow research grant EP/F044046/1.

** 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.

Despite its minimal syntax, logic programming can express recursive and even corecursive computations; it is adaptable to natural language processing; and it allows implicit parallel execution of programs. These features led to its various applications in computer science, such as *constraint logic programming*, *DATALOG*, and *parallel logic programming*. Logic programming has had direct and indirect impact on various disciplines of computer science.

There have been several successful attempts to give a categorical characterisation of logic programs and computations by logic programs. Among the earliest results was the characterisation of the first-order language as a *Lawvere theory* [2, 5, 6, 17], and most general unifiers (mgus) as *equalisers* [4] and *pullbacks* [6, 2]. There were several approaches to operational semantics of logic programming, notably, built upon the observation that logic programs resemble transition systems or rewriting systems; [5, 7].

The coalgebraic semantics we propose will ultimately rely upon Lawvere theories, mgus as equalisers and operational behavior of SLD-derivations given by state transitions. However, our original contribution is a coalgebraic characterisation of different derivation strategies in logic programming. As we show in Section 3, the algebraic semantics for logic programming [2, 6, 17] fails to give an account of the possibly infinite derivations that arise in the practice of logic programming. The coalgebraic semantics we propose fills this gap. Among the major advantages of this semantics is that it neatly models different strategies for parallel execution of logic programs, as we explain in Section 5.

In order to concentrate on derivations, rather than Lawvere theories, we ignore variables here. Ground logic programs have the advantage of yielding a variety of parallel derivation strategies [12, 23], as opposed to the general case, for which the algorithms of unification and SLD resolution are P-complete [27, 15]. The variable-free setting is more general than it may first appear: for some logic programs, only finitely many "ground" substitutions are possible; the exception being logic programs that describe potentially infinite data. Therefore, one often can emulate an arbitrary logic program by a variable-free one.

For the rest of the paper we assume that all clauses are ground, and that there are only finitely many predicate symbols, as is 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: p sends each atomic formula A to the set of the sets 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 $P_f P_f$-coalgebra on Set. In fact, we can go further. If we let $C(P_f P_f)$ be the cofree comonad on $P_f P_f$, then given a logic program qua $P_f P_f$-coalgebra, the corresponding $C(P_f P_f)$-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 5.

A similar analysis of SLD-derivations in logic programming can be given in terms of $P_f List$-coalgebras. Such an 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 $P_f P_f$- and $P_f List$-coalgebras, is that they model different strategies of parallel SLD-derivations.

The paper is organised as follows. In Section 2, we discuss the operational semantics for logic programs given by SLD-resolution. In Section 3 we analyse the finite and infinite SLD derivations that arise in the practice of logic programming, and briefly outline the existing approaches to coinductive logic programming. In Section 4, we show how to model variable-free logic programs as coalgebras and exhibit the role of the cofree comonad $C(P_f P_f)$; we discuss the difference between SLD-refutations modelled by $C(P_f P_f)$- and $C(P_f List)$-coalgebras. In Section 5, we show that the $C(P_f List)$-coalgebra is a sound and complete semantics for *and-or parallel SLD derivations*. We prove that semantics given by $C(P_f List)$-coalgebra is correct with respect to the *Theory of Observables* [7] for logic programming. In Section 6, we conclude the paper and discuss future work to be done.

2 Logic Programs and SLD Derivations

In this section, we recall the essential constructs of logic programming, as described, for instance, in Lloyd's standard text [19]. We start by describing the syntax of logic programs, after which we outline approaches to declarative and operational semantics of logic programs.

Definition 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. *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 indices $x_1, x_2, x_3, \ldots$.

Definition 3. *A* substitution *is a map* $\theta : Ter(\Sigma) \to Ter(\Sigma)$ *which satisfies*

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

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 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 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 [27, 15]. 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 a restriction yields easier implementations, because there is no need for unification algorithms, and such programs also can be implemented using parallel algorithms. We will return to this discussion in Section 4.

Remark 1. 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 show that both approaches are equally sound in case of ground logic programs, however, the first-order case requires the use of lists; and the majority of PROLOG interpreters treat the goals as lists.

One of our running examples will be the following program.

Example 1. Let GC (for graph connectivity) denote the logic program

```
connected(x,x) ←
connected(x,y) ← 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 the 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 $\leftarrow$ `connected(a,x)`.

Traditionally, logic programming has been given *least fixed point* semantics [19]. 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 [19]. A non-ground alternative to the ground fixed point semantics has been given in [9]; and further developed in terms of categorical logic [2, 6].

The fact that logic programs can be naturally represented via fixed point semantics has led to the development of *logic programs as inductive definitions*, [22, 14], 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 3, and 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 A_m.

Definition 6. *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 7. *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 $\leftarrow (A_1, \ldots, A_{m-1}, B_1, \ldots, B_q, A_{m+1}, \ldots, A_k)\theta$.*

A clause C_i^* is a *variant* of the corresponding clause C_i, if $C_i^* = C_i\theta$, with θ being a variable renaming substitution, such that variables in C_i^* do not appear in the derivation up to G_{i-1}. This process of renaming variables is called *standardising the variables apart.*

Definition 8. *An* SLD-derivation *of $P \cup \{G\}$ consists of a sequence of goals $G = G_0, G_1, \ldots$ called* resolvents*, a sequence $C_1, C_2, \ldots$ of variants 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 latter called proof-trees for their close relation to proof-trees in e.g. sequent calculus.

Definition 9. *Let P be a logic program and G be a goal. An* SLD-tree *for $P \cup \{G\}$ is a tree T satisfying the following:*

1. *Each node of the tree is a (possibly empty) goal.*
2. *The root node is G.*
3. *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 $C_1 = A^1 \leftarrow B_1^1, \ldots, B_q^1$, ..., $C_n = A^n \leftarrow B_1^n, \ldots, B_r^n$ 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*
$$\leftarrow (A_1, \ldots, A_{k-1}, B_1^i, \ldots, B_q^i, A_{k+1}, \ldots, A_m)\theta_i$$
4. *Nodes which are the empty clause have no children.*

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

Definition 10. *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 the 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 [26], 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 2. 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)
```

Figure 1 shows a proof-tree and an SLD-tree 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 [19, 9] show that every atom in the set computed by the least fixed point of T_P has a finite SLD-refutation, and vice versa.

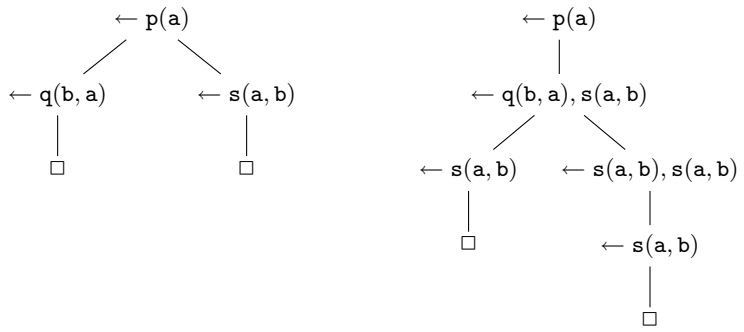

Fig. 1. A proof tree (left) and an SLD-tree (right) for a logic program of Example 2.

3 Finite and Infinite Computations by Logic Programs

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

Example 3. Consider the logic program from Example 1. It is easy to facilitate infinite SLD-derivations, by simply adding a clause that makes the graph cyclic: `edge(c,a)` ← . Taking a query ← `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 above, some derivations compute infinite objects.

Example 4. The following program `stream` defines the infinite stream of binary bits:

$$\texttt{bit}(0) \leftarrow$$
$$\texttt{bit}(1) \leftarrow$$
$$\texttt{stream(cons (x,y))} \leftarrow \texttt{bit(x)}, \texttt{stream(y)}$$

Programs like `stream` can be given declarative semantics via the *greatest* fixed point of the semantic operator T_P. However the fixed point semantics is incomplete in general [19]: it fails for some infinite derivations.

Example 5. The program below will be characterised by the greatest fixed point of the T_P operator, which 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, [14, 19, 22, 24]. But, the infinite SLD derivations of both finite and infinite objects have not yet received a uniform semantics, see Figure 2.

In [17, 18] we described algebraic fibrational semantics and proved soundness and completeness result for it with respect to finite SLD-refutations, see Figure 2. Alternative algebraic semantics for logic programming were given in [2, 6]. In this paper, we give a coalgebraic treatment of both finite and infinite SLD derivations, and prove soundness and completeness of this semantics for ground logic programs.

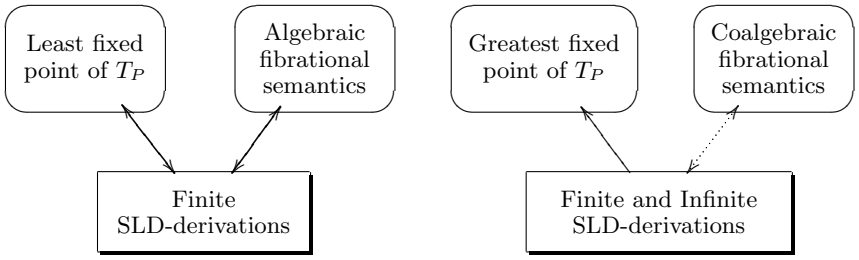

Fig. 2. Alternative declarative semantics for finite and infinite SLD-derivations. The solid arrows $\leftrightarrow$ show the semantics that are both sound and complete, and the solid arrow $\rightarrow$ indicates sound incomplete semantics, and the dotted arrow indicates the sound and complete semantics for ground logic programs we propose here.

4 Coalgebraic Semantics for Ground Logic Programs

In this section, we consider a coalgebraic semantics for ground logic programs. This semantics is intended to give meaning to both finite and infinite SLD-derivations.

Example 6. A ground logic program equivalent to the program from Example 1 can be obtained by taking all ground instances of non-ground clauses, such as, for example

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

When a non-ground program contains no function symbols (cf. Example 1), 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.

Variable-free logic programs and SLD derivations over such programs bear a strong resemblance to finitely branching transition systems (see also [5, 7]): atoms play the role of states, and the implication arrow $\leftarrow$ used to define clauses plays the role of the transition relation. 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 finite powerset functor P_f is iterated twice.

Proposition 1. *For any set At, there is a bijection between the set of variable-free logic programs over the set of atoms At and the set of P_fP_f-coalgebra structures on At.*

Proof. 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 $p : At \longrightarrow P_f(P_f(\mathrm{At}))$ sends an atom A to the set of bodies of those clauses in P with head A, each body being viewed as the set of atoms that appear in it.

Remark 2. One can alternatively view the bodies of clauses as lists of atoms. In this case, Proposition 1 can alternatively be given by P_fList-coalgebra. That is, 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, viewed as lists of atoms.

This coalgebraic representation of a logic program affords not only a representation of the program itself, but also of derivation trees for the program. As we now show, 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. This bears comparison with the greatest fixed point semantics for logic programs, see e.g. [19], but we do not pursue that here. The following theorem uses the construction given in [16, 28].

Theorem 1. *Given an endofunctor $H : Set \longrightarrow Set$ with a rank, the forgetful functor $U : H\text{-}Coalg \longrightarrow Set$ has a right adjoint R.*

Proof. R is constructed as follows. Given $Y \in Set$, we define a transfinite sequence of objects as follows. 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 $Y_1 = Y \times HY \xrightarrow{\pi_1} Y$. For a limit ordinal, let $Y_\alpha = \lim_{\beta<\alpha}(Y_\beta)$, determined by the sequence

$$Y_{\beta+1} \xrightarrow{\delta_\beta} Y_\beta.$$

If H has a rank, there exists α such that Y_α is isomorphic to $Y \times HY_\alpha$. This Y_α forms the cofree coalgebra on Y.

Corollary 1. *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*

$$G\text{-}Coalg \cong H\text{-}Coalg,$$

where G-Coalg is the category of G-coalgebras for the comonad G.

It will be helpful to make the correspondence between H-coalgebras and G-coalgebras more explicit. Given an H-coalgebra $p : Y \longrightarrow HY$, we construct maps $p_\alpha : Y \longrightarrow Y_\alpha$ for each ordinal α as follows. The map $p_0 : Y \longrightarrow Y$ is the identity, and for a successor ordinal, $p_{\alpha+1} = \langle id, Hp_\alpha \circ p\rangle : Y \longrightarrow Y \times HY_\alpha$. For limit ordinals, p_α is given by the appropriate limit. By definition, the object GY is given by Y_α for some α, and the corresponding p_α is the required G-coalgebra.

We can apply the general results given by the proof of Theorem 1 and Corollary 1 to extend our analysis of logic programs viewed as $P_f P_f$-coalgebras.

Construction 1 *Taking $p : At \longrightarrow P_f P_f(At)$, by the proof of Theorem 1, the corresponding $C(P_f P_f)$-coalgebra where $C(P_f P_f)$ is the cofree comonad on $P_f P_f$ is given as follows: $C(P_f P_f)(At)$ is given by a limit of the form*

$$\ldots \longrightarrow At \times P_f P_f(At \times P_f P_f(At)) \longrightarrow At \times P_f P_f(At) \longrightarrow At.$$

This chain has length ω. As above, we inductively define the objects $At_0 = At$ and $At_{n+1} = At \times P_f P_f At_n$, and the cone

$$p_0 = id : At \longrightarrow At (= At_0)$$
$$p_{n+1} = \langle id, P_f P_f(p_n) \circ p\rangle : At \longrightarrow At \times P_f P_f At_n (= At_{n+1})$$

and the limit determines the required coalgebra $\overline{p} : At \longrightarrow C(P_f P_f)(At)$.

Construction 2 *Note that by Remark 2, a similar construction for $P_f List$-coalgebras can be used to give semantics to SLD derivations for logic programs whose bodies are treated as lists of atoms: one just has to use $P_f List$ instead of $P_f P_f$ in construction 1.*

Construction 1 describes a structure that resembles the 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 10 and 9.

Example 7. Consider the logic program from Example 2.

The program has three atoms, namely `q(b,a)`, `s(a,b)` and `p(a)`. So $At = \{\texttt{q(b,a)}, \texttt{s(a,b)}, \texttt{p(a)}\}$. And the program can be identified with the $P_f P_f$-coalgebra structure on At given by
$p(\texttt{q(b,a)}) = \{\{\}, \{\texttt{s(a,b)}\}\}$, 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 the parallel refutation of `p(a)` depicted on the left side of Figure 3. Note that the nodes of the tree alternate between those labelled by atoms and those labelled by bullets ($\bullet$). The set of children of each bullet represents a goal, made up of the conjunction of the atoms in the labels. An atom with multiple children is the head of multiple clauses in the program: its children represent these clauses. We use the traditional notation $\Box$ to denote $\{\}$.

Where an atom has a single $\bullet$-child, we can elide that node without losing any information; the result of applying this transformation to our example is shown on the right in Figure 3. As we shall shortly explain, the resulting tree is precisely the *parallel and-or derivation tree* for the atomic goal $\leftarrow$ `p(a)`; see Definition 11.

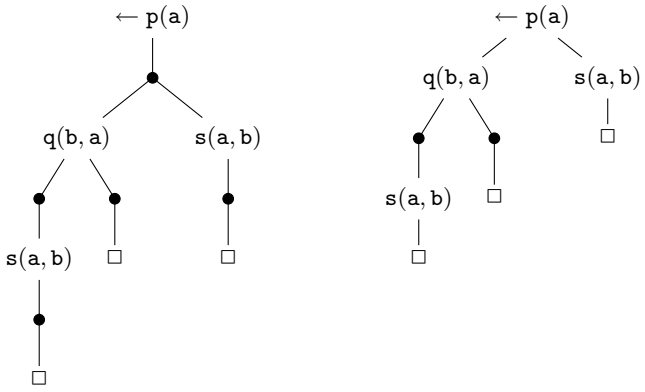

Fig. 3. The action of $\overline{p} : \mathrm{At} \longrightarrow C(P_f P_f)(\mathrm{At})$ on `p(a)`, and the corresponding and-or derivation tree.

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 [`q(b,a) :: s(a,b)`].

The action of the corresponding $C(P_f List)$-coalgebra on $\mathtt{p(a)}$ can be depicted similarly to Fig. 3, subject to a mildly modified understanding of the picture. In replacing P_f by $List$, the order of atoms in a goal is vital, and an atom may occur more than once. So to account for $List$, we must regard the diagram as a tree with a given embedding of the children of each $\bullet$-node in the plane, and allow several children of the same $\bullet$-node to be labelled by the same atom. Thus the children of a $\bullet$-node are understood as a list, corresponding to their relative locations in the plane, and the same atom may appear more than once among them. On the other hand, the $\bullet$-nodes are not ordered, so the children of the atom-nodes are not embedded in the plane.

5 Coalgebraic semantics and parallel execution of logic programs

The tree shown in Example 7 differs from the SLD-tree (cf. Definition 9) and the proof-tree (cf. Definition 10) constructed for the same logic program in Example 2. The reason is that the derivations modelled by the G-coalgebras have strong relation to *parallel logic programming*, [27, 15], 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 three main types of parallelism used in implementations of logic programs are *and-parallelism* and *or-parallelism*, and their combination; see [12, 23] for an excellent analysis 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 9 and Example 2. Or-parallelism has successfully been exploited in Prolog in the Aurora [20] 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 10 and Example 2. Independent And-parallelism has been successfully exploited in the &-Prolog System [13].

The comonad we have constructed in the previous section models a synthetic form of parallelism: *And-Or parallelism*. The most common way to express And-Or parallelism in logic programs is through *and-or trees* [12], 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 in the Andorra system [8], with many more implementations following it [12].

Definition 11. *Let P be a logic program and $G =\leftarrow A$ be an atomic goal. The* parallel and-or derivation tree *for A is the possibly infinite tree T satisfying the following properties.*

- *A is the 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 $B^i_1\theta_i, \ldots, B^i_n\theta_i$.*

There are non-trivial choices about when to regard two trees as equivalent: one possibility is when they are equivalent in the plane, another is when they are combinatorially equivalent, a third is a hybrid. These correspond to characterisations as $C(ListList)$-coalgebras, as $C(P_fP_f)$-coalgebras, and as $C(P_fList)$-coalgebras respectively, yielding three theorems, the second of which is as follows.

Theorem 2 (Soundness and completeness). *Let P be a variable-free logic program, At the set of all atoms appearing in P, and $\bar{p} : At \longrightarrow C(P_fP_f)(At)$ the $C(P_fP_f)$-coalgebra generated by P. (Recall that $\bar{p}$ is constructed as a limit of a cone p_n over an ω-chain.) Then, for any atom A, $\bar{p}(A)$ expresses precisely the same information as that given by the parallel and-or derivation tree for A, that is, the following holds:*

- *For a derivation step n of the parallel and-or tree for A, $p_n(A)$ is isomorphic to the and-or parallel tree for A of depth n.*
- *The and-or tree for A is of finite size and has the depth n iff $\bar{p}(A) = p_n(A)$.*
- *The and-or tree for A is infinite iff $\bar{p}(A)$ is given by the element of the limit $\lim_\omega(p_n)(At)$ of an infinite chain given by Construction 1.*

Proof. We use Constructions 1 and 2 and put, for every $A \in At$:

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

The limit of the sequence is precisely the structure described by Construction 1, moreover, for each A in At, $p_0(A)$ corresponds to the root of the and-or parallel tree, and each $p_n(A)$ corresponds to the nth parallel derivation step for A.

Note that the result above relates to the *Theory of Observables* for logic programming developed in [7]. According to this theory, traditional characterisation of logic programs in terms of input/output behaviour is not sufficient for the purposes of program analysis and optimisation. Therefore, it is useful to have complete information about the SLD-derivation, e.g., the sequences of goals, most general unifiers, and variants of clauses. The following four observables are the most important for the theory [10, 7].

- *Partial answers* are the substitutions associated to a resolvent in any SLD-derivation; *correct partial answers* are substitutions associated to a resolvent in any SLD-refutation.
- *Call patterns* are atoms selected in any SLD-derivation; *correct call patterns* are atoms selected in any SLD-refutation.
- *Computed answers* are the substitutions associated to an SLD-refutation.
- *A successful derivation* is observation of successful termination.

As argued in [10, 7], one of the main purposes of giving a semantics to logic programs is its ability to observe equal behaviors of logic programs and distinguish logic programs with different computational behavior. Therefore, the choice of observables and semantic models is closely related to the choice of equivalence relation defined over logic programs; [10].

Definition 12. *Let P_1 and P_2 be ground logic programs. Then we define $P_1 \approx P_2$ if and only if, for any (not necessarily ground) goal G, the following four conditions hold:*

1. *G has a refutation in P_1 if and only if G has a refutation in P_2;*
2. *G has the same set of computed answers in P_1 and P_2.*
3. *G has the same set of (correct) partial answers in P_1 and P_2.*
4. *G has the same set of call patterns in P_1 and P_2.*

Following the terminology of [10, 7], we can state the following *correctness result*.

Theorem 3. *For ground logic programs P_1 and P_2, if the parallel and-or tree for P_1 is equal to the parallel and-or tree for P_2, then $P_1 \approx P_2$.*

The converse of Theorem 3 does not hold. That is, there can be observationally equivalent programs that have different and-or parallel trees.

Example 8. Consider two logic programs, P_1 and P_2, whose clauses are exactly the same, with the exception of one clause: P_1 contains $A \leftarrow B_1, \ldots, B_i, \texttt{false}, \ldots, B_n$; and P_2 contains the clause $A \leftarrow B_1, \ldots B_i, \texttt{false}$ instead. The atoms in the clauses are such that $B_1, \ldots, B_i$ have refutation in P_1 and P_2, and `false` is an atom that has no refutation in the programs. In this case, all derivations that involve the two clauses in P_1 and P_2 will always fail on `false`, and P_1 will be observationally equivalent to P_2. However, their and-or parallel trees will give an account to all the atoms in the clause.

6 Conclusions

In this paper, we have modelled the derivation strategies of logic programming by coalgebra, with the bulk of our work devoted to modelling variable-free programs. We plan to extend the coalgebraic analysis to non-ground logic programs. In general, and-or parallelism characterised here by $P_f P_f$-coalgebras and $P_f List$-coalgebras is not sound for first-order derivations. In practice of logic programming, a more sophisticated parallel derivations are used, in order to coordinate substitutions computed in parallel. For example, *composition (and-or parallel) trees* were introduced in [12] as a way to simplify the implementation of the traditional and-or trees, which required synchronisation of substitutions in the branches of parallel derivation trees. In the coalgebraic semantics of first-order logic programs, $P_f List$-coalgebras will play a more prominent role than $P_f P_f$-coalgebras, and their relation to composition trees will become important.

The other topic for future investigations will be to explore how different approaches to *bisimilarity*, as e.g. analysed in [25], relate to the observational semantics [7] and observational equivalence of logic programs. In Theorem 3, we used the equality of parallel and-or trees to characterise observational equivalence of programs, but one could consider bisimilarity of $C(P_f List)$-coalgebras instead. The choice of bismilarity relation will determine a particular kind of the observational equivalence characterised by the semantics. This line of reserach will become more prominent in case of non-ground logic programs.

The situation regarding higher-order logic programming languages such as λ-*PROLOG* [21] is more subtle. Despite their higher-order nature, such logic programming 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.

Another area of research would be to investigate the operational meaning of coinductive logic programming [3, 11, 24] which requires a slight modification to the algorithm of SLD-resolution we have considered in this paper.

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. G. Amato, J. Lipton, and R. McGrail. On the algebraic structure of declarative programming languages. *Theor. Comput. Sci.*, 410(46):4626–4671, 2009.
3. D. Ancona, G. Lagorio, and E. Zucca. Type inference by coinductive logic programming. In *TYPES 2008*, volume 5497 of *LNCS*, pages 1–18. Springer, 2009.
4. A. Asperti and S. Martini. Projections instead of variables: A category theoretic interpretation of logic programs. In *ICLP*, pages 337–352, 1989.
5. F. Bonchi and U. Montanari. Reactive systems, (semi-)saturated semantics and coalgebras on presheaves. *Theor. Comput. Sci.*, 410(41):4044–4066, 2009.
6. R. Bruni, U. Montanari, and F. Rossi. An interactive semantics of logic programming. *TPLP*, 1(6):647–690, 2001.
7. M. Comini, G. Levi, and M. C. Meo. A theory of observables for logic programs. *Inf. Comput.*, 169(1):23–80, 2001.

8. 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.
9. M. Falaschi, G. Levi, C. Palamidessi, and M. Martelli. Declarative modeling of the operational behavior of logic languages. *TCS*, 69(3):289–318, 1989.
10. M. Gabrielli, G. Levi, and M. Meo. Observable behaviors and equivalnences of logic programs. *Information and Computation*, 122(1):1–29, 1995.
11. G. Gupta, A. Bansal, R. Min, L. Simon, and A. Mallya. Coinductive logic programming and its applications. In *ICLP 2007*, volume 4670 of *LNCS*, pages 27–44. Springer, 2007.
12. 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.
13. M. V. Hermenegildo and K. J. Greene. &-prolog and its performance: Exploiting independent and-parallelism. In *ICLP*, pages 253–268, 1990.
14. M. Jaume. On greatest fixpoint semantics of logic programming. *J. Log. Comput.*, 12(2):321–342, 2002.
15. P. C. Kanellakis. Logic programming and parallel complexity. In *Foundations of Deductive Databases and Logic Programming.*, pages 547–585. M. Kaufmann, 1988.
16. 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.
17. Y. Kinoshita and A. J. Power. A fibrational semantics for logic programs. In *Proceedings of the Fifth International Workshop on Extensions of Logic Programming*, volume 1050 of *LNAI*. Springer, 1996.
18. E. Komendantskaya and J. Power. Fibrational semantics for many-valued logic programs: Grounds for non-groundness. In *JELIA*, volume 5293 of *LNCS*, pages 258–271, 2008.
19. J. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 2nd edition, 1987.
20. 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.
21. D. Miller and G. Nadathur. Higher-order logic programming. In *ICLP*, pages 448–462, 1986.
22. L. C. Paulson and A. W. Smith. Logic programming, functional programming, and inductive definitions. In *ELP*, pages 283–309, 1989.
23. E. Pontelli and G. Gupta. On the duality between or-parallelism and and-parallelism in logic programming. In *Euro-Par*, pages 43–54, 1995.
24. 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.
25. S. Staton. Relating coalgebraic notions of bisimulation. In *Proceedings of 3rd Conference on Algebra and Coalgebra in Computer Science (CALCO'09)*, volume 5728 of *LNCS*, pages 191–205, 2009.
26. L. Sterling and E. Shapiro. *The art of Prolog*. MIT Press, 1986.
27. J. D. Ullman and A. V. Gelder. Parallel complexity of logical query programs. *Algorithmica*, 3:5–42, 1988.
28. J. Worrell. Toposes of coalgebras and hidden algebras. *Electr. Notes Theor. Comput. Sci.*, 11, 1998.