

# Coalgebraic Derivations in Logic Programming\*

Ekaterina Komendantskaya<sup>1</sup> and John Power<sup>2</sup>

- 1 Department of Computing,  
University of Dundee, UK  
katya@computing.dundee.ac.uk
- 2 Department of Computer Science,  
University of Bath, UK  
A.J.Power@bath.ac.uk

---

## Abstract

Coalgebra may be used to provide semantics for SLD-derivations, both finite and infinite. We first give such semantics to classical SLD-derivations, proving results such as adequacy, soundness and completeness. Then, based upon coalgebraic semantics, we propose a new sound and complete algorithm for parallel derivations. We analyse this new algorithm in terms of the Theory of Observables, and we prove soundness, completeness, correctness and full abstraction results.

**1998 ACM Subject Classification** D.1.6 Logic Programming; F.3.2 Semantics of Programming Languages; F.1.2 Models of Computation

**Keywords and phrases** Logic programming, SLD-resolution, coalgebra, Lawvere theories, coinductive logic programming, concurrent logic programming

**Digital Object Identifier** 10.4230/LIPIcs.CSL.2011.352

## 1 Introduction

In the standard formulations of logic programming, such as in Lloyd's book [19], a first-order logic program  $P$  consists of a finite set of clauses of the form  $A \leftarrow A_1, \dots, A_n$ , where  $A$  and the  $A_i$ 's are atomic formulae, typically containing free variables, and where  $A_1, \dots, A_n$  is understood to mean the conjunction of the  $A_i$ 's: note that  $n$  may be 0.

A running example of a logic program in this paper is as follows.

► **Example 1.1.** Let ListNat denote the logic program

```
nat(0) ←
nat(s(x)) ← nat(x)
list(nil) ←
list(cons x y) ← nat(x), list(y)
```

The program involves variables  $x$  and  $y$ , function symbols  $0$ ,  $s$ ,  $nil$  and  $cons$ , and predicate symbols  $nat$  and  $list$ , with the choice of notation designed to make the intended meaning of the program clear.

SLD-resolution, which is a central algorithm for logic programming, takes a goal  $G$ , typically written as  $\leftarrow B_1, \dots, B_n$ , where the list of  $B_i$ 's is again understood to mean a

---

\* We acknowledge EPSRC PDRF EP/F044046/2; and the SICSA distinguished visiting fellowship.



conjunction of atomic formulae, typically containing free variables, and constructs a proof for an instantiation of  $G$  from substitution instances of the clauses in  $P$  [19]. The algorithm uses Horn-clause logic, with variable substitution determined universally to make a selected atom in  $G$  agree with the head of a clause in  $P$ , then proceeding inductively. Section 2 recalls the various definitions.

SLD-resolution is sound and complete with respect to least fixed point semantics [19]. But the analysis afforded by least fixed point operators pertains only to finite SLD derivations, whereas infinite SLD derivations are also common in the practice of programming. An example is as follows.

► **Example 1.2.** The following program `Stream` defines the infinite stream of binary bits:

```

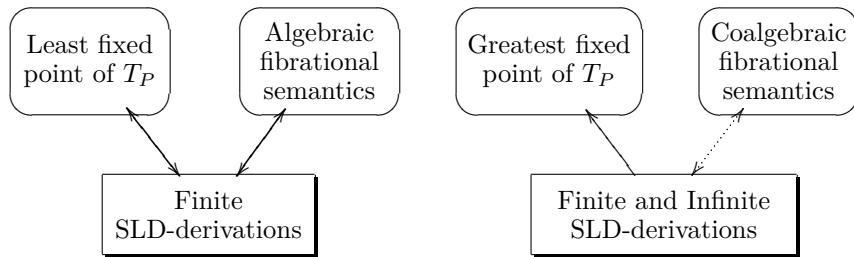
bit(0) ←
bit(1) ←
stream(scons (x,y)) ← bit(x),stream(y)
    
```

Programs like `Stream` can be given declarative semantics via the *greatest* fixed point of the semantic operator  $T_P$ , see also Section 2. But greatest fixed point semantics is incomplete in general [19] as it fails for some infinite derivations.

► **Example 1.3.** The program  $R(x) \leftarrow R(f(x))$  is characterised by the greatest fixed point of the  $T_P$  operator, which contains  $R(f^\omega(a))$ , but no infinite term is computed by SLD-resolution.

There have been numerous attempts to resolve the mismatch between infinite derivations and greatest fixed point semantics, e.g., [2, 11, 13, 19, 20, 22, 25]. But infinite SLD derivations of both finite and infinite objects have not yet received a uniform semantics, see Figure 1.

In [15, 17], we described an algebraic (fibrational) semantics for logic programming and proved soundness and completeness results for it with respect to SLD-resolution. Other forms of algebraic semantics for logic programming have been given in [1, 5]. Here, we give coalgebraic semantics for both finite and infinite SLD derivations, and prove soundness and completeness results for it, see Sections 3, 4. That constitutes the first main contribution of the paper.



■ **Figure 1** Alternative declarative semantics for finite and infinite SLD-derivations. The arrows  $\leftrightarrow$  show the semantics that are both sound and complete, and the arrow  $\rightarrow$  indicates sound incomplete semantics. The dotted arrow indicates the sound and complete semantics we propose here.

Another distinguishing feature of logic programming languages is that they allow implicit parallel execution of programs. The three main types of parallelism used in implementations are *and-parallelism*, *or-parallelism*, and their combination: see [12, 23] for analysis.

*Or-parallelism* arises when more than one clause unifies with the goal: 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 has been exploited in Aurora and Muse, both of which have shown good speed-up results over a considerable range of applications.

*And-parallelism* arises when more than one atom is present in the goal. That is, given a goal  $G = \leftarrow B_1, \dots, B_n$ , an *and-parallel algorithm* for SLD resolution looks for SLD derivations for each  $B_i$  simultaneously, subject to the condition that the atoms must not share variables. Such cases are known as *independent and-parallelism*. Independent and-parallelism has been successfully exploited in &-PROLOG.

The coalgebraic models we discuss in this paper exhibit a synthetic form of parallelism: *and-or parallelism*. The most common way to express and-or parallelism in logic programs is via *and-or trees* [12], which consist of both *or-nodes* and *and-nodes*. And-or parallel PROLOG works best for variable-free logic programs or DATALOG, and was first implemented in Andorra [7], see also [12]. But many first-order algorithms are P-complete and hence inherently sequential [8, 14]. This especially concerns first-order unification and variable substitution in the presence of variable dependencies. So extensions of and-or parallel derivations to the general case require complicated algorithms that coordinate variable substitution in different branches of and-or parallel derivation trees [12]. If such synchronisation is omitted, parallel SLD-derivations may lead to unsound results, see also Section 5.

In Section 5, we propose an alternative derivation algorithm inspired by our coalgebraic semantics [18]. It inherently models substitutions in a uniform way, so that additional techniques for synchronisation of substitutions are not required. We support the algorithm with soundness, completeness, correctness and full abstraction results with respect to the coalgebraic semantics. That is the second major contribution of the paper.

The underlying category theory of this paper was developed in [18], but the relationship with ordinary logic programming syntax was not systematically developed there, in particular with none of the syntax/semantics results given there.

## 2 First-order logic programming

We recall some basic definitions from [19].

A *signature*  $\Sigma$  consists of a set of *function symbols*  $f, g, \dots$  each equipped with a fixed *arity*. The arity of a function symbol is a natural number indicating the number of its arguments. Nullary (0-ary) function symbols are allowed: these are called *constants*. Given a countably infinite set  $Var$  of variables, the set  $Ter(\Sigma)$  of *terms* over  $\Sigma$  is defined inductively:  $x \in Ter(\Sigma)$  for every  $x \in Var$ . If  $f$  is an  $n$ -ary function symbol ( $n \geq 0$ ) and  $t_1, \dots, t_n \in Ter(\Sigma)$ , then  $f(t_1, \dots, t_n) \in Ter(\Sigma)$ . Variables will be denoted  $x, y, z$ , sometimes with indices  $x_1, x_2, x_3, \dots$ . A *substitution* is a map  $\theta : Ter(\Sigma) \rightarrow Ter(\Sigma)$  which satisfies  $\theta(f(t_1, \dots, t_n)) \equiv f(\theta(t_1), \dots, \theta(t_n))$  for every  $n$ -ary function symbol  $f$ .

We define an *alphabet* to consist of a signature  $\Sigma$ , the set  $Var$ , and a set of *predicate symbols*  $P, P_1, P_2, \dots$ , each assigned an arity. Let  $P$  be a predicate symbol of arity  $n$  and  $t_1, \dots, t_n$  be terms. Then  $P(t_1, \dots, t_n)$  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.

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

Given a first-order language  $\mathcal{L}$ , a *logic program* consists of a finite set of clauses of the form  $A \leftarrow A_1, \dots, A_n$ , where  $A, A_1, \dots, A_n$  ( $n \geq 0$ ) are atoms. The atom  $A$  is called the *head* of a clause, and  $A_1, \dots, A_n$  is called its *body*. Clauses with empty bodies are called *unit clauses*. A *goal* is given by  $\leftarrow B_1, \dots, B_n$ , where  $B_1, \dots, B_n$  ( $n \geq 0$ ) are atoms.

Traditionally, logic programming has been modelled by *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  $T_P(I)$  on  $2^{B_P}$  by sending  $I$  to the set  $\{A \in B_P : A \leftarrow A_1, \dots, A_n \text{ is a ground instance of a clause in } P \text{ with } \{A_1, \dots, A_n\} \subseteq I\}$ . 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 this semantics was further developed in terms of categorical logic in [1, 5].

The fact that logic programs can be represented naturally by least fixed point semantics led to the development of *logic programs as inductive definitions* [22, 13]. Operational semantics for logic programs is given by SLD-resolution, a goal-oriented proof-search procedure.

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

► **Definition 2.1.** Let a goal  $G$  be  $\leftarrow A_1, \dots, A_m, \dots, A_k$  and a clause  $C$  be  $A \leftarrow B_1, \dots, B_q$ . Then  $G'$  is *derived* from  $G$  and  $C$  using mgu  $\theta$  if the following conditions hold:

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

A clause  $C_i^*$  is a *variant* of the clause  $C_i$  if  $C_i^* = C_i\theta$ , with  $\theta$  being a variable renaming substitution such that variables in  $C_i^*$  do not appear in the derivation up to  $G_{i-1}$  (see the notation below). This process of renaming variables is called *standardising the variables apart*; we assume it throughout the paper without explicit mention.

► **Definition 2.2.** An *SLD-derivation* of  $P \cup \{G\}$  consists of a sequence of goals  $G = G_0, G_1, \dots$  called *resolvents*, a sequence  $C_1, C_2, \dots$  of variants of program clauses of  $P$ , and a sequence  $\theta_1, \theta_2, \dots$  of mgus such that each  $G_{i+1}$  is derived from  $G_i$  and  $C_{i+1}$  using  $\theta_{i+1}$ . An *SLD-refutation* of  $P \cup \{G\}$  is a finite SLD-derivation of  $P \cup \{G\}$  that has the empty clause  $\square$  as its last goal. If  $G_n = \square$ , we say that the refutation has length  $n$ . The composition  $\theta_1, \theta_2, \dots$  is called *computed answer*.

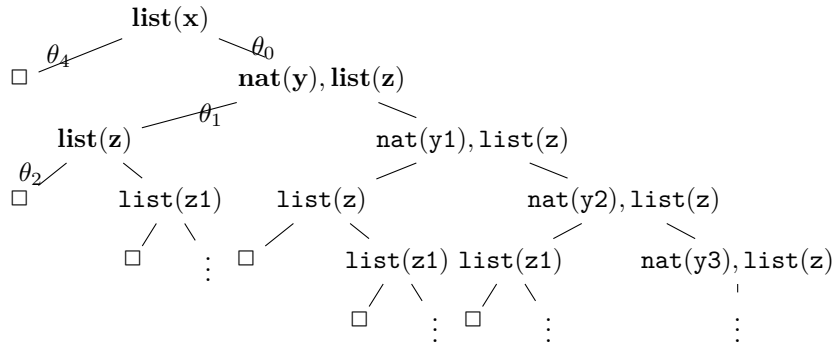
SLD-resolution is P-complete, and hence inherently sequential [8]. Operationally, SLD-derivations can be characterised by *SLD-trees*.

► **Definition 2.3.** 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  $\leftarrow A_1, \dots, A_m$ ,  $m > 0$  is a node in  $T$ , and it has  $n$  children, then there exists  $A_k \in A_1, \dots, A_m$  such that  $A_k$  is unifiable with exactly  $n$  distinct clauses  $C_1 = A^1 \leftarrow B_1^1, \dots, B_q^1, \dots, C_n = A^n \leftarrow B_1^n, \dots, B_q^n$  in  $P$  via mgus  $\theta_1, \dots, \theta_n$ , and, for every  $i \in \{1, \dots, n\}$ , the  $i$ th child node is given by the goal

$$\leftarrow (A_1, \dots, A_{k-1}, B_1^i, \dots, B_q^i, A_{k+1}, \dots, A_m)\theta_i$$

4. nodes which are the empty clause have no children.



■ **Figure 2** An SLD-tree for ListNat with the goal  $\leftarrow list(x)$ . A possible computed answer is given by the composition of  $\theta_0 = x/cons(y, z)$ ,  $\theta_1 = y/0$ ,  $\theta_2 = z/nil$ ; Another computed answer is  $\theta_4 = x/nil$ .

► **Example 2.4.** Figure 2 shows an SLD-tree for ListNat (Example 1.1). Note that a similar goal  $stream(x)$  in the logic program Stream from Example 1.2 will produce a very different SLD-tree in that it will not have leaf nodes. The nodes will infinitely alternate between  $stream(x)$  and  $bit(y), stream(z)$ , modulo variable renaming.

SLD-resolution is sound and complete with respect to least fixed point semantics. The classical theorems of soundness and completeness of this operational semantics [19] show that every atom in the set computed by the least fixed point of  $T_P$  has a finite SLD-refutation, and vice versa.

### 3 Coalgebraic Semantics for SLD-derivations

Logic programs resemble, and indeed induce, transition systems or rewrite systems, hence coalgebras. That fact has been used to study their operational semantics, e.g., in [4, 6]. In [16], we developed the idea for variable-free logic programs, extending it to first-order programs in [18]. In this section, we recall the relevant details.

Given a set  $At$  of atoms, there is a bijection between the set of variable-free logic programs over  $At$  and the set of  $P_f P_f$ -coalgebra structures on  $At$ , i.e., functions  $p : At \rightarrow P_f P_f(At)$ , where  $P_f$  is the finite powerset functor: each atom of a logic program  $P$  is the head of finitely many clauses, and the body of each of those clauses contains finitely many atoms.

The endofunctor  $P_f P_f$  necessarily has a cofree comonad  $C(P_f P_f)$  on it as follows.

► **Proposition 3.1.** Let  $C(P_f P_f)$  denote the cofree comonad on  $P_f P_f$ . For any set  $At$ ,  $C(P_f P_f)(At)$  is the limit of a diagram of the form

$$\dots \rightarrow At \times P_f P_f(At \times P_f P_f(At)) \rightarrow At \times P_f P_f(At) \rightarrow At.$$

Given  $p : At \rightarrow P_f P_f(At)$ , put  $At_0 = At$  and  $At_{n+1} = At \times P_f P_f(At_n)$ , and consider the cone defined inductively as follows:

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

The limiting property determines the coalgebra  $\bar{p} : At \rightarrow C(P_f P_f)(At)$ .

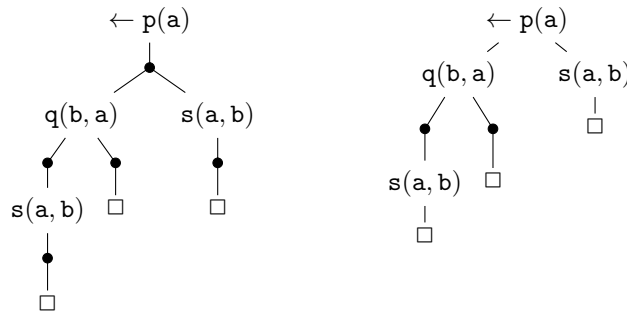
The main result of [16] asserted that if  $C(P_f P_f)$  is the cofree comonad on  $P_f P_f$ , then, given a logic program  $P$ , the induced  $C(P_f P_f)$ -coalgebra structure characterises the parallel and-or derivation trees (cf. [12]) of  $P$ .

► **Example 3.2.** Consider the variable-free logic program:  $q(b, a) \leftarrow; s(a, b) \leftarrow; p(a) \leftarrow q(b, a), s(a, b); q(b, a) \leftarrow s(a, b)$ .

The program has three atoms, namely  $q(b, a)$ ,  $s(a, b)$  and  $p(a)$ . So  $At = \{q(b, a), s(a, b), p(a)\}$ . 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.  $p(s(a, b)) = \{\{\}\}$ , i.e., the one element set consisting of the empty set.  $p(p(a)) = \{\{q(b, a), s(a, b)\}\}$ .

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 labeled by atoms and those labeled 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  $\square$  to denote  $\{\}$ .

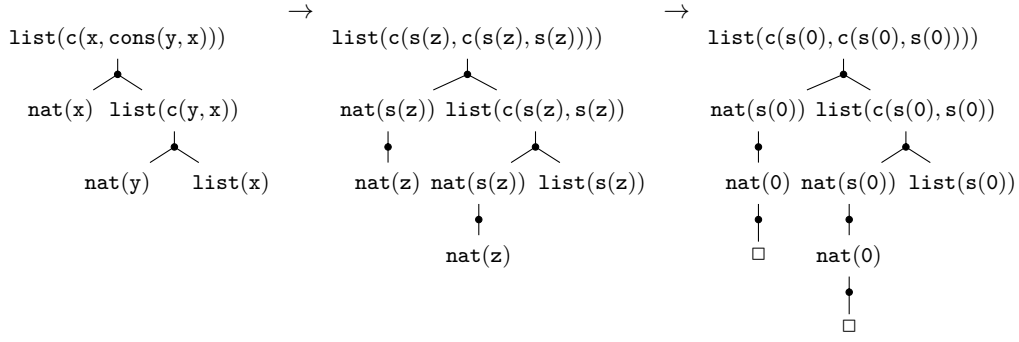
Where an atom has a single  $\bullet$ -child, we can elide that node without losing any information; the result of applying this transformation to our example is shown on the right in Figure 3. The resulting tree is precisely the parallel and-or derivation tree [12] for the atomic goal  $\leftarrow p(a)$ .



■ **Figure 3** The action of  $\bar{p} : At \rightarrow C(P_f P_f)(At)$  on  $p(a)$ , and the corresponding parallel and-or derivation tree [12].

In [18], we extend this to first-order logic programs using *Lawvere theories*, cf. [1, 4, 5, 15], modelling most general unifiers (mgu's) by *equalisers*. cf. [3]: given a signature  $\Sigma$ , the Lawvere theory  $\mathcal{L}_\Sigma$  generated by  $\Sigma$  has objects given by natural numbers and maps from  $n$  to  $m$  given by equivalence classes of substitutions  $\theta$  of  $m$  variables by terms generated by the function symbols in  $\Sigma$  applied to  $n$  variables. We shall shortly give an example; for formal definitions and theorem see [18].

Given a logic program  $P$  with function symbols in  $\Sigma$ , we extend the set  $At$  of atoms in a variable-free logic program to the functor from  $\mathcal{L}_\Sigma^{op}$  to  $Set$  sending a natural number  $n$  to the set  $At(n)$  of atomic formulae with at most  $n$  variables generated by the predicate symbols in  $P$ . One can extend any endofunctor  $H$  on  $Set$  to the endofunctor  $[\mathcal{L}_\Sigma^{op}, H]$  on  $[\mathcal{L}_\Sigma^{op}, Set]$  that sends  $F : \mathcal{L}_\Sigma^{op} \rightarrow Set$  to the composite  $HF$ . We would like to model  $P$  by the putative  $[\mathcal{L}_\Sigma^{op}, P_f P_f]$ -coalgebra  $p : At \rightarrow P_f P_f At$  that, at  $n$ , takes an atomic formula  $A(x_1, \dots, x_n)$  with at most  $n$  variables, considers all substitutions of clauses in  $P$  whose head agrees with  $A(x_1, \dots, x_n)$ , and gives the set of sets of atomic formulae in antecedents, mimicking the construction for variable-free logic programs.



■ **Figure 4** The left hand tree represents  $\bar{p}(\text{list}(\text{cons}(x, \text{cons}(y, x))))$  and the second tree represents  $\bar{p}At((s, s))(\text{list}(\text{cons}(x, \text{cons}(y, x))))$ , i.e.,  $\bar{p}(\text{list}(\text{cons}(s(z), \text{cons}(s(z), s(z))))$ , and the tree on the right depicts  $\bar{p}At(0)At((s, s))(\text{list}(\text{cons}(x, \text{cons}(y, x))))$ ;  $\text{cons}$  is abbreviated by  $c$ .

In fact, to make the theory work, we need to extend *Set* to *Poset*, natural transformations to *lax natural transformations*, and replace the outer instance of  $P_f$  by  $P_c$  - the countable powerset functor (as recursion generates countability). Subject to those replacements,  $p : At \rightarrow P_c P_f At$  behaves as above, giving a  $Lax(\mathcal{L}_\Sigma^{op}, P_c P_f)$ -coalgebra structure on  $At$ . Extending Proposition 3.1,  $p$  determines a  $Lax(\mathcal{L}_\Sigma^{op}, C(P_c P_f))$ -coalgebra structure  $\bar{p} : At \rightarrow C(P_c P_f)(At)$ .

► **Example 3.3.** Consider ListNat as in Example 1.1. Suppose we start with  $A(x, y) \in At(2)$  given by the atomic formula  $\text{list}(\text{cons}(x, \text{cons}(y, x)))$ . Then  $\bar{p}(A(x, y))$  is the element of  $C(P_c P_f)At(2)$  expressible by the tree on the left hand side of Figure 4.

The coalgebraic structure means any substitution, whether determined by an mgu or not, applies to the whole tree. The lax naturality means a substitution potentially yields two different trees: one given by substitution into the tree, then pruning to remove redundant branches, the other given by substitution into the root, then applying  $\bar{p}$ .

For example, we can substitute  $s(z)$  for both  $x$  and  $y$  in  $\text{list}(\text{cons}(x, \text{cons}(y, x)))$ . This substitution is given by applying  $At$  to the map  $(s, s) : 1 \rightarrow 2$  in  $\mathcal{L}_\Sigma$ . So  $At((s, s))(A(x, y))$  is an element of  $At(1)$ . Its image under  $\bar{p}(1) : At(1) \rightarrow C(P_c P_f)At(1)$  is the element of  $C(P_c P_f)At(1)$  expressible by the tree on the right hand side of Figure 4. The laxness of the naturality of  $\bar{p}$  is indicated by the increased length, in two places, of the second tree. Observe that, before those two places, the two trees have the same structure: that need not always be exactly the case, as substitution in a tree could involve pruning if substitution instances of two different atoms yield the same atom.

Now suppose we make the further substitution of 0 for  $z$ . This substitution is given by applying  $At$  to the map  $0 : 0 \rightarrow 1$  in  $\mathcal{L}_\Sigma$ . In Figure 4, we depict  $\bar{p}(0)At(0)At((s, s))(A(x, y))$  on the right. Two of the leaves of the latter tree are labeled by  $\square$ , but one leaf, namely  $\text{list}(s(0))$  is not, so the tree does not yield a proof. Again, observe the laxness.

The trees shown in Example 3.3 differ from the corresponding SLD-tree determined by Definition 2.3. The main reason for this is that the derivations modelled by the coalgebraic semantics have strong relation to *parallel logic programming*, [26, 14], while SLD-trees describe sequential derivation strategies.

In following sections, we shall show how our coalgebraic semantics relates to sequential derivations, and how it can be used to introduce a new concurrent derivation algorithm.



## 4 Coalgebraic Semantics and Infinite Derivations

In this section, we state formally the theorems that relate the coalgebraic semantics of the previous section to first-order (possibly infinite) derivations in logic programming. We start by introducing a special kind of derivation tree that is suitable for representing derivations described by the coalgebraic semantics.

► **Definition 4.1.** Let  $P$  be a logic program and  $G = \leftarrow A$  be an atomic goal. The *coinductive derivation tree* for  $A$  is a 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 and-node  $A'$  occurring in  $T$ , there exist exactly  $m > 0$  distinct clauses  $C_1, \dots, C_m$  in  $P$  (a clause  $C_i$  has the form  $B_i \leftarrow B_1^i, \dots, B_{n_i}^i$ , for some  $n_i$ ), such that  $A' = B_1\theta_1 = \dots = B_m\theta_m$ , for some substitutions  $\theta_1, \dots, \theta_m$ , then  $A'$  has exactly  $m$  children given by or-nodes, such that, for every  $i \in m$ , the  $i$ th or-node has  $n_i$  children given by and-nodes  $B_1^i\theta_i, \dots, B_{n_i}^i\theta_i$ .

► **Example 4.2.** Examples of coinductive derivation trees are given in Figures 3 and 4.

Note that, comparing this with the SLD-resolution algorithm and the corresponding SLD-trees, the definition of coinductive derivation tree restricts unification to the case of *term matching*, i.e., the substitution  $\theta$  unifying atoms  $A_1$  and  $A_2$  is applied only to one atom, e.g.  $A_1 = A_2\theta$ , whereas traditionally mgus satisfy  $A_1\theta = A_2\theta$ . The term-matching algorithm is parallelisable, in contrast to the unification algorithm, which is inherently sequential [8].

We define the *depth of a coinductive tree* inductively as follows. The root of a coinductive tree has depth 0. For an and-node  $x$ , if its immediate parent and-node has depth  $d$ , then  $x$  has depth  $d + 1$ . The depth of a tree is defined to be the depth of its deepest branch.

For all the running examples we use in this paper, there will be only one coinductive tree for every goal. However, this will not be the case for programs containing clauses in which not all the variables appearing in the body appear in the head.

► **Example 4.3.** In [16] we analyse the program determining whether two nodes in a graph are connected. It contains the clause `connected(x, y) ← edge(x, z), connected(z, y)`, note the appearance of  $z$ .

According to Definition 4.1 such clauses may induce a family of coinductive trees - as there can be a countable number of substitutions  $\theta'_i, \dots, \theta''_i$  that match a given goal with the clause  $C_i$ , each of these substitutions differing only with respect to assignment to  $z$ .

► **Definition 4.4.** Let  $P$  be a logic program and  $G = \leftarrow A$  be an atomic goal. The *coinductive forest*  $F$  for  $A$  is a set of all coinductive derivation trees for  $A$ . We say that the forest has *depth*  $n$  if the deepest tree in  $F$  has length  $n$ . A coinductive forest  $F$  has *breadth*  $k$  if at most  $k$  distinct variables appear in all and-nodes of all of its trees together.

► **Theorem 4.5 (Adequacy).** For any logic program  $P$  and for any atom  $A$  generated by the predicate symbols of  $P$  and  $k$  distinct variables  $x_1, \dots, x_k$ ,  $\bar{p}(k)(A)$  expresses precisely the same information as that given by a coinductive forest  $F$  for the goal  $A$ . That is, the following holds:

- $p_n(k)(A)$  is isomorphic to the coinductive forest of depth  $n$  and breadth  $k$ .
- $F$  has the finite depth  $n$  if and only if  $\bar{p}(k)(A) = p_n(k)(A)$ .



- $F$  has infinite depth if and only if  $\bar{p}(k)(A)$  is given by the element of the limit of the infinite chain given by (the extension of) Proposition 3.1.

**Proof.** By (the extension of) Proposition 3.1, for every atomic formula  $A$ :

- $p_0(k)(A) = A$
- $p_1(k)(A) = (A, \{\{B^1\theta, \dots, B^m\theta\}, \text{ such that } B \leftarrow B^1, \dots, B^m \text{ is a clause in } P \text{ with } B\theta = A \text{ and } B^1\theta, \dots, B^m\theta \text{ have variables among } x_1, \dots, x_k.\})$
- $p_2(k)(A) = (A, \{\{(B^1\theta, \{\{C_1^1\theta_1, \dots, C_1^{m_1}\theta_1\}) \text{ such that } C \leftarrow C_1^1, \dots, C_1^{m_1} \text{ is a clause in } P \text{ with } C\theta_1 = B^1\}, \dots \text{ and } C_1^1\theta_1, \dots, C_1^{m_1}\theta_1 \text{ have variables among } x_1, \dots, x_k.\})\})$
- etc.

The limit of the sequence is precisely (the extension of) the structure described by Proposition 3.1. For each atomic formula  $A$ ,  $p_0(k)(A)$  corresponds to the root of a coinductive derivation tree, and, more generally, each  $p_n(k)(A)$  corresponds to the coinductive forest of breadth  $k$ , as far as depth  $n$ . ◀

► **Example 4.6.** Infinite coinductive trees arise in programs similar to that in Example 1.3. The infinite tree arising from this program contains a chain of alternating  $\bullet$ 's and atoms  $R(x)$ ,  $R(f(x))$ ,  $R(f(f(x)))$ , etcetera. Note that infinite terms are not nodes of the tree. Programs like **Stream** and **ListNat** in Examples 1.1 and 1.2, do not give rise to infinite coinductive derivation trees, see Figures 4 and 6. But they do give rise to infinite SLD-trees, see Figure 2. This is because substitution, determined by term-matching, is applied only to clauses, and not to goals, when a coinductive derivation tree is built. Infinite derivations in these programs may be modelled by infinite chains of derivation trees.

We can express Theorem 4.5 in terms of a traditional-style soundness and completeness result that relates the semantics to SLD-refutations. For this purpose, we define *success subtrees* of coinductive derivation trees, as follows.

► **Definition 4.7.** Let  $P$  be a logic program,  $A$  be a goal, and  $T$  be the coinductive derivation tree determined by  $P$  and  $A$ . A subtree  $T'$  of  $T$  is called a *success subtree* of  $T$  if it satisfies the following conditions:

- the root of  $T'$  is the root of  $T$ ;
- if an and-node belongs to  $T'$ , and the node has  $k$  children in  $T$  given by or-nodes, only one of these or-nodes belongs to  $T'$ .
- if an or-node belongs to  $T'$ , then all its children given by and-nodes in  $T$  belong to  $T'$ .
- all the leaves of  $T'$  are and-nodes represented by  $\square$ .

► **Theorem 4.8** (Soundness and Completeness of SLD-resolution relative to coinductive derivation trees.). *Let  $P$  be a logic program, and  $G$  be a goal.*

1. *Soundness. If there is an SLD-refutation for  $G$  in  $P$  with computed answer  $\theta$ , then there exists a coinductive derivation tree for  $G\theta$  that contains a success subtree.*
2. *Completeness. If a coinductive derivation tree for  $G\theta$  contains a success subtree, then there exists an SLD-refutation for  $G$  in  $P$ , with computed answer  $\lambda$  such that there exists substitution  $\sigma$  such that  $\lambda = \sigma\theta$ .*

**Proof.** The proof is given by induction on the length of the SLD-refutations and the depth of the coinductive trees. Part 2 also requires some analysis of computed answers. If a program does not contain clauses similar to Example 4.3, then  $\sigma$  is an identity substitution or a variable renaming, otherwise  $\sigma$  is determined by all the substitutions computed by the SLD-derivations that involved assigning terms to the variables appearing in the body but not the head of clauses in  $P$ . ◀

► **Corollary 4.9** (Soundness and Completeness of SLD-resolution relative to coalgebraic semantics). *Given a logic program  $P$ , SLD-refutations in  $P$  are sound and complete with respect to the  $\text{Lax}(\mathcal{L}_\Sigma^{\text{op}}, P_c P_f)$ -coalgebra determined by  $P$ .*

**Proof.** Follows from Theorems 4.5 and 4.8. ◀

Our coalgebraic analysis relates to the *Theory of Observables* for logic programming developed in [6]. In that theory, the traditional characterisation of logic programs in terms of input/output behavior and successful derivations is not sufficient for the purposes of program analysis and optimisation. One requires more complete information about SLD-derivations, e.g., the sequences of goals, most general unifiers, and variants of clauses. Moreover, infinite derivations can be meaningful. The following four observables are the most important for the theory [9, 6].

- **Definition 4.10. 1.** *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.
- 2. *Call patterns* are atoms selected in any SLD-derivation; *correct call patterns* are atoms selected in any SLD-refutation.
- 3. *Computed answers* are the substitutions associated to an SLD-refutation.
- 4. *A successful derivation* is the observation of successful termination.

As argued in [9, 6], a key goal of semantics to logic programs is to observe equal behavior of logic programs and to distinguish logic programs with different computational behavior. The choice of observables and semantic models is closely related to the choice of equivalence relation defined over logic programs [9].

► **Definition 4.11.** Let  $P_1$  and  $P_2$  be logic programs. Put  $P_1 \approx P_2$  if and only if, for a 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  $P_1$  as in  $P_2$
- 3.  $G$  has the same set of (correct) partial answers in  $P_1$  as in  $P_2$
- 4.  $G$  has the same set of call patterns in  $P_1$  as in  $P_2$ .

Using the terminology of [9, 6], we can state the following *correctness result* that relates the traditional sequential SLD-derivations of Section 2 to our coalgebraic semantics. In the next theorem, we assume that there is a common algorithm that assigns terms to variables appearing only in the bodies of clauses as explained in Example 4.3.

► **Theorem 4.12** (Correctness). *For logic programs  $P_1$  and  $P_2$ , if for every atomic goal  $\leftarrow A$ , the coinductive forest for  $P_1$  and  $A$  is equal to the coinductive forest for  $P_2$  and  $A$ , then  $P_1 \approx P_2$ .*

The converse of Theorem 4.12, the *full abstraction result*, does not hold. That is, there can be observationally equivalent programs that have different coinductive derivation trees.

► **Example 4.13.** Consider the logic programs  $P_1$  and  $P_2$ , whose clauses are the same, with the exception of one clause:  $P_1$  contains  $A \leftarrow B_1, \dots, B_i, \text{false}, \dots, B_n$ ; and  $P_2$  contains the clause  $A \leftarrow B_1, \dots, B_i, \text{false}$  instead. The atoms in the clauses are such that  $B_1, \dots, B_i$  have refutations in  $P_1$  and  $P_2$ , and **false** is an atom that has no refutation in the programs. In this case, assuming a left-to-right sequential evaluation strategy, 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 coinductive derivation trees give account to all atoms in the clause.

The results of this section show that parallel trees arising from the coalgebraic semantics of Section 3 naturally model finite and infinite derivations. The nature of the failure of the full abstraction result suggests that the coalgebraic semantics of Section 3 more naturally supports concurrent computation, rather than sequential SLD-derivations. For this reason, we introduce a novel algorithm for concurrent derivations in the next section.

## 5 Applications in Concurrent Logic Programming

In this section, we exploit the concurrent nature of our coalgebraic semantics, equivalently coinductive derivation trees. Operationally, the major difference between coinductive trees and SLD-trees lies in the concurrent versus sequential modes of execution, which are crucial for the computation of call patterns, (correct) partial answers and soundness of computations.

We first consider a concurrent computational model already in the literature: and-or-parallel trees [12].

► **Definition 5.1.** [12] Let  $P$  be a logic program and let  $\leftarrow A$  be an atomic goal (possibly with variables). The *and-or parallel derivation tree* for  $A$  is the possibly infinite tree  $T$  satisfying the following properties.

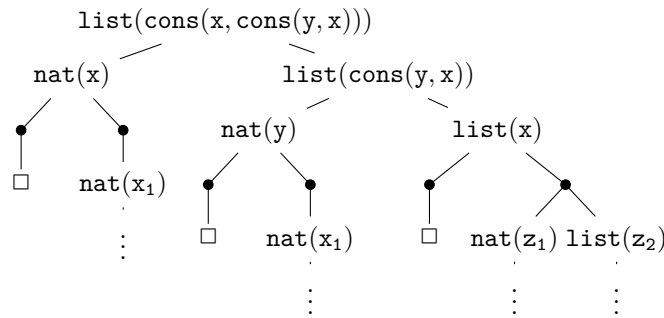
- $A$  is the root of  $T$ .
- Each node in  $T$  is either an and-node or an or-node.
- Each or-node is given by  $\bullet$ .
- Each and-node is an atom.
- For every node  $A'$  occurring in  $T$ , if  $A'$  is unifiable with only one clause  $B \leftarrow B_1, \dots, B_n$  in  $P$  with mgu  $\theta$ , then  $A'$  has  $n$  children given by and-nodes  $B_1\theta, \dots, B_n\theta$ .
- For every node  $A'$  occurring in  $T$ , if  $A'$  is unifiable with exactly  $m > 1$  distinct clauses  $C_1, \dots, C_m$  in  $P$  via mgu's  $\theta_1, \dots, \theta_m$ , then  $A'$  has exactly  $m$  children given by or-nodes, such that, for every  $i \in m$ , if  $C_i = B^i \leftarrow B_1^i, \dots, B_n^i$ , then the  $i$ th or-node has  $n$  children given by and-nodes  $B_1^i\theta_i, \dots, B_n^i\theta_i$ .

An example of an and-or tree is given in Figure 3. Example 3.2 demonstrates and [16] formally proves that coinductive trees and and-or trees produce the same results in the variable-free case. However, a naive extension of Definition 5.1 to the first-order case yields inconsistent derivations.

► **Example 5.2.** Figure 5 shows the and-or parallel tree that finds a refutation  $\theta = \{x/0, y/0, x/nil\}$  for the goal `list(cons(x, cons(y, x)))`, although this answer is not sound.

A solution proposed in [12] was given by *composition (and-or parallel) trees*. Construction of composition trees involves additional algorithms that synchronise substitutions in the branches of and-or trees. Composition trees contain a special kind of *composition nodes* used whenever both and- and or-parallel computations are possible for one goal. A composition node is a list of atoms in the goal. If, in a goal  $G = \leftarrow B_1, \dots, B_n$ , an atom  $B_i$  is unifiable with  $k > 1$  clauses, then the algorithm 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  $B_1, \dots, B_n$ , and  $n$  and-parallel edges. Thus, all possible combinations of all possible or-choices at every and-parallel step are given.

Here, we propose coinductive trees of Definition 4.1 as an alternative to composition trees. Comparing coinductive derivation trees with and-or trees, coinductive trees are more intrinsic: and-or parallel trees have most general unifiers built into a single tree, whereas,



■ **Figure 5** Unsound refutation by and-or parallel tree, with  $\theta = \{x/0, y/0, x/nil\}$ .

mgus determine only tree transformations for coinductive trees. Taking unification issues from the level of individual leaves to the level of trees affects computations at least in two ways. Parallel proof-search in branches of a coinductive tree does not require synchronisation of variables in different branches. Moreover, for programs that are guarded by constructors - such as `ListNat` and `Stream`, we avoid having infinite branches or infinite number of variables in a single tree. We shall illustrate with our leading example.

► **Example 5.3.** The coinductive trees from Figure 4 agree with the first part of the and-or parallel tree for `list(cons(x, cons(y, x)))` in Figure 5. But the coinduction tree has leaves `nat(x)`, `nat(y)` and `list(x)`, whereas the and-or tree follows those nodes, using substitutions determined by mgu's. Moreover, those substitutions need not be consistent with each other: not only are there two ways to unify each of `nat(x)`, `nat(y)` and `list(x)`, but also there is no consistent substitution for `x` at all. In contrast, the coinduction trees capture such cases.

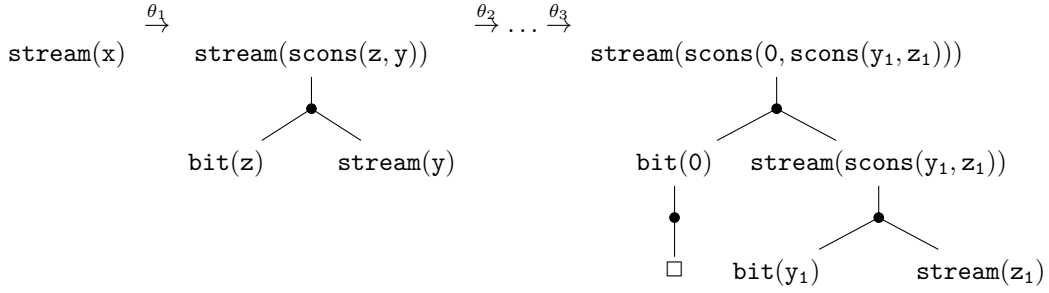
We can go further and introduce a new derivation algorithm that allows proof search using coinduction trees. We modify the definition of a goal by taking it to be a pair  $\langle A, T \rangle$ , where  $A$  is an atom, and  $T$  is the coinduction tree determined by  $A$ , as in Definition 4.1, in which we restrict the choice of substitutions  $\theta_1, \dots, \theta_m$  to the most general unifiers only, in which case  $T$  is uniquely determined by  $A$ .

► **Definition 5.4.** Let  $G$  be a goal given by an atom  $\leftarrow A$  and the coinductive tree  $T$  induced by  $A$ , and let  $C$  be a clause  $H \leftarrow B_1, \dots, B_n$ . Then goal  $G'$  is *coinductively derived* from  $G$  and  $C$  using mgu  $\theta$  if the following conditions hold:

- $A'$  is a leaf atom, called the *selected atom*, in  $T$ .
- $\theta$  is an *mgu* of  $A'$  and  $H$ .
- $G'$  is given by the atom  $\leftarrow A\theta$  and the coinduction tree  $T'$  determined by  $A\theta$ .

► **Definition 5.5.** A *coinductive derivation* of  $P \cup \{G\}$  consists of a sequence of goals  $G = G_0, G_1, \dots$  called *coinductive resolvents* and a sequence  $\theta_1, \theta_2, \dots$  of mgus such that each  $G_{i+1}$  is derived from  $G_i$  using  $\theta_{i+1}$ . A *coinductive refutation* of  $P \cup \{G\}$  is a finite coinductive derivation of  $P \cup \{G\}$  such that its last goal contains a success subtree. If  $G_n$  contains a success subtree, we say that the refutation has length  $n$ .

Coinductive derivations resemble *tree rewriting*. In applying SLD-derivation, one's primary interest lies in derivations of atomic goals. But in order to make the induction work, one must generalise goals from being atoms to being lists of atoms, see Definition 2.1. In coinductive tree, this information would be represented by a list of nodes in a truncation of



■ **Figure 6** Coinductive derivation of length 3 for the goal  $G = \text{stream}(x)$  and the program  $\text{Stream}$ , with  $\theta_1 = x/\text{cons}(z, y)$  and  $\theta_2 = z/0, \theta_3 = y/\text{cons}(y_1, z_1)$ .

the coinductive tree. To analyse coinductive derivations, we generalise the definition of a goal a little further, extending it from being an atom  $A$  to being the coinductive derivation tree for  $A$ , see Definition 4.1. For every goal  $G = \langle A, T \rangle$ , there can be several transitions to a new goal, and these transitions can be made concurrently.

► **Example 5.6.** Figure 6 shows a coinductive derivation of length 3 for the goal  $G = \text{stream}(x)$  and the program  $\text{Stream}$  from Example 1.2.

► **Theorem 5.7** (Soundness and Completeness of coinductive resolution relative to coalgebraic semantics.). *Let  $P$  be a program built over the signature  $\Sigma$ , and  $G = \langle A(\bar{t}), T \rangle$  be a goal.*

1. *Soundness. If there is a coinductive derivation of length  $n$  of  $P \cup \{G\}$  with an answer  $\theta = \theta_1 \circ \dots \circ \theta_n$ , and if  $G_n = \langle A_n(\bar{t}_n), T_n \rangle$ ,  $\bar{t}_n$  having  $k$  distinct variables, then  $\bar{t}_n = \bar{t}\theta$  and  $\bar{p}(k)(At(\theta))(A(\bar{t}))$  is isomorphic to the coinductive forest  $F$  of breadth  $k$  determined by  $A_n$ .*
2. *Completeness. Given the  $\text{Lax}(\mathcal{L}_\Sigma^{\text{op}}, P_c P_f)$ -coalgebra structure  $\bar{p}$  generated by  $P$ , let  $\theta$  be a map in  $\mathcal{L}_\Sigma^{\text{op}}$ , and let  $C$  be the structure determined by evaluating  $\bar{p} : At(\theta) \rightarrow C(P_c P_f)(At)$  at a natural number  $k$  and applying it to an atomic formula  $A(x_1, \dots, x_k)$ . Then there exists a derivation from  $G = \langle A(x_1, \dots, x_k), T \rangle$  to  $G_n = \langle A_n, T_n \rangle$ , with  $A_n = A(x_1, \dots, x_k)\sigma$ , such that there exists a substitution  $\rho$  such that  $\theta = \sigma\rho$  and the coinductive forest for  $A_n\rho$  is isomorphic to  $C$ .*

**Proof.** The proof proceeds by induction on the length of derivations, using the constructions of Theorem 4.5. ◀

Theorem 5.7 characterises all derivations, not only finite ones, although it can be restricted to coinductive refutations. In general, there are two levels of computation at which both infinity and concurrency can be implemented in coinductive derivations. One level is that of the coinduction trees given by the goals; and the second level is the transitions between the goals. Depending on the applications and resources for parallelisation, the coinductive derivation algorithm above offers several choices as follows.

Every coinductive tree in a goal is necessarily concurrent, but transitions between coinduction trees can be done in a sequential or a concurrent manner. That is, if there are several non-empty leaves in a tree, any such leaf can be unified with some clause in  $P$ . Such leaves can provide substitutions for sequential or concurrent tree transitions. In Figure 6, the substitution  $\theta' = \theta_2\theta_3$  is derived by considering mgus for two leaves in  $G_1 = \langle \text{stream}(\text{scons}(z, y)), T_1 \rangle$ ; but, although two separate and-leaves were used to compute  $\theta'$ ,  $\theta'$  was computed by composing the two substitutions sequentially, and only one

tree,  $T_3$ , was produced. However, we could concurrently derive two trees from  $T_2$  instead,  $G'_2 = \langle \text{stream}(\text{scons}(0, y)), T_2 \rangle$  and  $G''_2 = \langle \text{stream}(\text{scons}(z, \text{scons}(y_1, z_1))), T'_2 \rangle$ .

There are choices concerning how to treat infinite coinductive trees arising in derivations. As Example 4.6 shows, some definitions of infinite objects do not give rise to infinite coinduction trees, e.g., `Stream` gives rise to an infinite sequence of finite coinduction trees, cf. Figure 6. This applies equally to any (potentially) infinite data defined using *constructors*, such as `scons` in `Stream` or `cons` and `nil` in `ListNat`. So one may view infinite coinduction trees as “bad” cases, in which (co)recursion is *not guarded by constructors*. In this case, one might decide to halt any derivation of this kind, and amend the program before proceeding. Alternatively, one may decide to prune infinite branches, and continue to look for derivations in other or-branches for the same unchanged logic program.

Finally, as Figure 6 shows, coinductive programs such as `Stream` may give rise to infinite derivations of coinduction trees, in which case implementation may prune the chain of derivations as [11, 25] suggest, or, if infinite production of new streams is desirable, let the coinductive derivations run.

We can now remedy the *full abstraction* result that we have proven to fail for the SLD-derivations, see Section 5. We once again characterise coinductive derivations from the point of view of the Theory of Observables. In particular, we can routinely adapt Definitions 4.10 and 4.11 to coinductive derivations using substitutions and call patterns determined by coinductive derivations rather than by SLD-derivations. Then the following correctness and full abstraction results hold.

► **Theorem 5.8.**  $P_1 \approx P_2$  if and only if the  $\text{Lax}(\mathcal{L}_\Sigma^{\text{op}}, P_c P_f)$ -coalgebra structure generated by  $P_1$  is equivalent to the  $\text{Lax}(\mathcal{L}_\Sigma^{\text{op}}, P_c P_f)$ -coalgebra structure generated by  $P_2$ .

**Proof. (Sketch.)** Proof proceeds by induction on constructions described in Theorems 4.5 and 5.7. ◀

## 6 Conclusions and Further Work

The analysis of this paper can be extended to more expressive logic programming languages, such as [10, 24, 21], also to functional programming languages in the style of [22, 2]. We deliberately chose our running examples to correspond to definitions of inductive or coinductive types in such languages.

The key fact driving our analysis has been the observation that the implication  $\leftarrow$  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 [10, 24]. 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  $\lambda$ -*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.

---

### References

- 1 G. Amato, J. Lipton, and R. McGrail. On the algebraic structure of declarative programming languages. *Theor. Comput. Sci.*, 410(46):4626–4671, 2009.
- 2 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.

- 3 A. Asperti and S. Martini. Projections instead of variables: A category theoretic interpretation of logic programs. In *ICLP*, pages 337–352, 1989.
- 4 F. Bonchi and U. Montanari. Reactive systems, (semi-)saturated semantics and coalgebras on presheaves. *Theor. Comput. Sci.*, 410(41):4044–4066, 2009.
- 5 R. Bruni, U. Montanari, and F. Rossi. An interactive semantics of logic programming. *TPLP*, 1(6):647–690, 2001.
- 6 M. Comini, G. Levi, and M. C. Meo. A theory of observables for logic programs. *Inf. Comput.*, 169(1):23–80, 2001.
- 7 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.
- 8 C. Dwork, P.C. Kanellakis, and J.C. Mitchell. On the sequential nature of unification. *Journal of Logic Programming*, 1:35–50, 1984.
- 9 M. Gabrielli, G. Levi, and M.C. Meo. Observable behaviors and equivalences of logic programs. *Information and Computation*, 122(1):1–29, 1995.
- 10 J.-Y. Girard. Linear logic. *Theor. Comput. Sci.*, 50:1–102, 1987.
- 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.
- 13 M. Jaume. On greatest fixpoint semantics of logic programming. *J. Log. Comput.*, 12(2):321–342, 2002.
- 14 P. C. Kanellakis. Logic programming and parallel complexity. In *Foundations of Deductive Databases and Logic Programming.*, pages 547–585. M. Kaufmann, 1988.
- 15 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.
- 16 E. Komendantskaya, G. McCusker, and J. Power. Coalgebraic semantics for parallel derivation strategies in logic programming. In *Proc. of AMAST'2010 - 13th Int. Conf. on Algebraic Methodology and Software Technology*, volume 6486 of *LNCS*, 2010.
- 17 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.
- 18 E. Komendantskaya and J. Power. Coalgebraic semantics for derivations in logic programming. In *CALCO'11*, 2011.
- 19 J.W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 2nd edition, 1987.
- 20 Zoran Majki?. Coalgebraic semantics for logic programming. In *18th Workshop on (Constraint) Logic Programming, WLP 2004, March 04-06*, 2004.
- 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 D.J. Pym. *The Semantics and Proof Theory of the Logic of Bunched Implications*, volume 26 of *Applied Logic Series*. Kluwer Academic Publishers, 2002.
- 25 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.
- 26 J.D. Ullman and A.V. Gelder. Parallel complexity of logical query programs. *Algorithmica*, 3:5–42, 1988.